INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Intraprocedural Pointer Analysis for Container-Centric Applications

Albert Cohen — Peng Wu — David Padua

N° 4289

Octobre 2001

THÈME 4

ISSN 0249-6399 ISRN INRIA/RR--4289--FR+ENG

ROCQUENCOURT

Intraprocedural Pointer Analysis for Container-Centric Applications

Albert Cohen * , Peng Wu † , David Padua †

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet A3

Rapport de recherche n° 4289 — Octobre 2001 — 32 pages

Abstract: As programmers look forward to designing high performance applications with object-oriented models, compilers must support higher-level analyses and optimizations. Pointer analysis for container-centric applications is one of these: it exploits abstract semantics of container structures (e.g., lists, trees, associative maps) provided by standard libraries and toolkits. Extending shape analysis work by Sagiv, Reps and Wilhelm, we capture aliasing properties through dedicated points-to graphs. Formalization in abstract interpretation allowed us to prove the abstraction's and transfer functions' safety. We ran the analysis on small examples. It achieved precise memory disambiguations useful to parallelization and optimization.

Key-words: points-to graph, abstract container type, abstract interpretation, alias and shape analysis

* A3 Project, INRIA Rocquencourt
† DCS, University of Illinois at Urbana-Champaign

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Analyse de pointeurs intraprocédurale pour les applications manipulant des conteneurs

Résumé : Les programmeurs s'intéressent de plus en plus aux modèles orientés-objets pour la conception d'applications à hautes performances ; les compilateurs doivent ainsi réaliser des analyses et des optimisations de plus haut niveau. Une de ces analyses s'intéresse aux pointeurs dans les applications manipulant des conteneurs : elle exploite la sémantique abstraite des structures de conteneurs (listes, arbres, tables associatives...) proposées par des bibliothèques et des boîtes à outils standard. En étendant l'analyse de forme (*shape analysis*) de Sagiv, Reps et Wilhelm, nous décrivons les relations d'alias à l'aide de graphes *points-to* spécifiques. Une preuve de correction des abstractions et des fonctions de transfert a été réalisée dans le formalisme de l'interprétation abstraite. L'analyse a été expérimentée sur de courts exemples. Elle permet de séparer précisément les références mémoire, mettant ainsi en évidence une parallélisation et des optimisations intéressantes.

Mots-clés : graphe *points-to*, type abstrait de conteneur, interprétation abstraite, analyse d'alias et de forme

Contents

1 Introduction

Object-oriented design plays an increasing role in performance-critical codes. The shift toward higher level design models enables software reuse, generic programming, and aids the construction of large software systems. When dealing with general-purpose programs, arrays should share their preeminence with more general *container* components, such as lists, hash-tables, sets, and so on. These general-purpose containers are often provided by standard libraries, such as C++ Standard Template Library (STL) and Java Standard Development Kit (SDK).

This work aims at the automatic retrieval of pointer properties in the context of container classes. Besides classical points-to relations, we are especially interested in the *connection* between containers and their elements. Several intuitions were borrowed from [23]: pointer analysis should not limit itself to *pointer-chasing*, i.e., recursive dereferences of the form `p = p.next` (in Java syntax). Indeed, object-oriented design tends to hide such technicalities behind *containers* and *iterators* that traverse them. Although implementations of container classes could be very complex, especially with hand optimizations and generic codings, they often exhibit clean semantics through their programming interfaces. Thus, for a better analyzability of object-oriented codes, the key is to exploit semantics of critical classes in compilers.

The paper presents a points-to analysis for container traversals. The presentation uses Java syntax and semantics (Java references are indeed "well-behaved" pointers), but the general framework is not limited to Java.[1] Studying real-world kernels, we show that our technique achieves a very high precision as soon as container traversals expose enough regularity. For some of the kernels, the same precision cannot be achieved by classical techniques applied to an equivalent pointer-chasing implementation. The result of the analysis can be interpreted as shape [11, 19], alias [3, 8] or connectivity [10]. These properties can further improve analyses such as dependence tests and various optimizations.

We also address cost issues of such a high-precision analysis. In fact, when efficiency matters, one may still be very precise about container operations while applying inexpensive rules on other pointer assignments. This is a clear benefit of using standard containers as opposed to hand-written pointer-chasing implementations: heuristics can be made conscious of the level of precision required, without additional "hints" from the programmer or the "blind" degradation of k-limiting.

Section 2 presents the motivation for this work and introduces the main concepts. Section 4 describes the points-to graph and how to abstract the connection between a container and its elements. The points-to algorithm is given in Section 5. More general alias and shape analysis applications are presented in Section 7. Section 6 applies our technique to illustrative examples. Section 8 discusses possible improvements. We then compare our approach with others in Section 9. Section 10 concludes and sketches future work. All technical proofs have been moved to Section B in the appendix.

2 Overview of the Method

The analysis targets containers that can be traversed as linear lists, e.g., lists and vectors. We only studied a small set of basic container operations: `first()` and `last()` generates a new iterator attached to the container; `advance()` and `retreat()` move an iterator; `get()`, `put()`, `insert()` and `delete()` update/accesses a container through an iterator. The syntax of these operations is borrowed from the Java Generic Library (JGL).[2] The semantics is slightly simplified, e.g., no implicit iterator move in container accesses.

Let's consider the example in Figure 1 and suppose that `list` refers to a *non-circular* list. We would like to infer whether `a` and `b` are aliased at point 6. This would depend on two conditions: whether iterator `i` indicates distinct positions at points 2 and 4; and, whether distinct objects are attached to distinct positions inside `list`. The first condition can be decided by tracking "moves" of an iterator. The second condition is related to the "shape" of a container. A container is a *comb* when *distinct elements are attached at distinct positions*. *Combness* is the container counterpart of classical shape attributes, e.g., *listness* or *treeness*. Figure 1 gives both examples of `list` pointing to a *comb* or *not comb* container: `a` and `b` are unaliased in the first graph, and aliased in the second one. Moreover, after executing `i.put(o)`, one would like to replace

[1] Extension to C/C++ should be possible using well-known techniques [16, 21, 2].

[2] A project from ObjectSpace, see `http://www.objectspace.com`.

```
1  Foo o = new Object ();
2  Iterator i = list.first ();
3  Foo a = i.get ();
4  i.advance ();
5  Foo b = i.get ();
6  i.put (o);
```

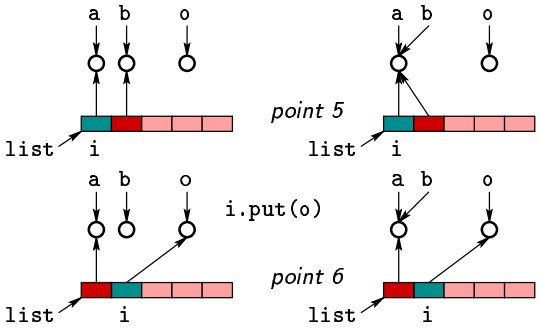


Figure 1: Motivating example

the element referenced by `i` with the object referenced by `o`. This implies killing the connection between `list` and the object referenced by `b`.

In this paper, combness is used as an intermediate property for capturing points-to, aliasing and connectivity relations. Nevertheless, it might also be applied directly to parallelization (combined with dependence analysis) [23] and data structure layout optimizations, such as object inlining [9] (combness is silimar to so called, one-to-one fields) and "flattening" of Java arrays [22].

Overall, the analysis is based on a graph representation that abstract the points-to relation between pointers. Our technique is strongly influenced by Sagiv, Reps and Wilhelm's shape analysis for destructive updating [18]. In addition, container-specific information is encoded into the graph, such as iterator attachment, combness and container positions. Focusing on precision, the analysis is flow-sensitive and handles destructive updates of the form `v.f = null` (for general pointers) or `i.delete()` (for containers). We also introduce a new scheme to abstract element retrieval through `get()`. It is a generalization of Sagiv, Reps and Wilhelm's *materialization* [18].

3 The Direction Abstraction

The points-to analysis needs to model precisely the connection between containers and the objects they store. This section introduces the *direction* abstraction to capture properties of such connections.

3.1 Connections and Directions

Our points-to graphs uses special container edges to model the link between containers and the objects attached. We call the concrete state abstracted by such container edge a *connection*. In particular, objects abstracted by the source/target nodes of the edge are called the *source containers*/*target objects* of the connection. A connection has two important properties, the *domain* and the *injectivity*,

- The *domain* of a connection describes at which positions the target objects of the connection are attached at its source containers.
- The *injectivity* of a connection tells whether each target object of the connection is attached at a single position of the source containers.

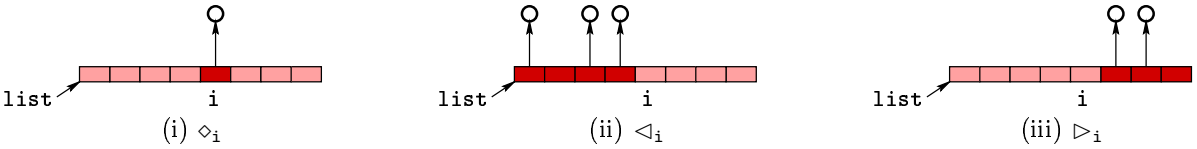
We use *directions* to describe the domain and injectivity of a connection. In the points-to graph, this is reflected as labeling container edges by directions. First, let us define *primitive directions*, examples are given in Figure 2.

$\diamond_{\texttt{i}}$ describes any injective connection whose domain is the *single* position of (indicated by) `i`;

$\lhd_{\texttt{i}}$ describes any injective connection whose domain *precedes* the position of `i`;

$\rhd_{\texttt{i}}$ describes any injective connection whose domain *follows* the position of `i`;

$\varnothing$ describes any connection whose domain is empty;



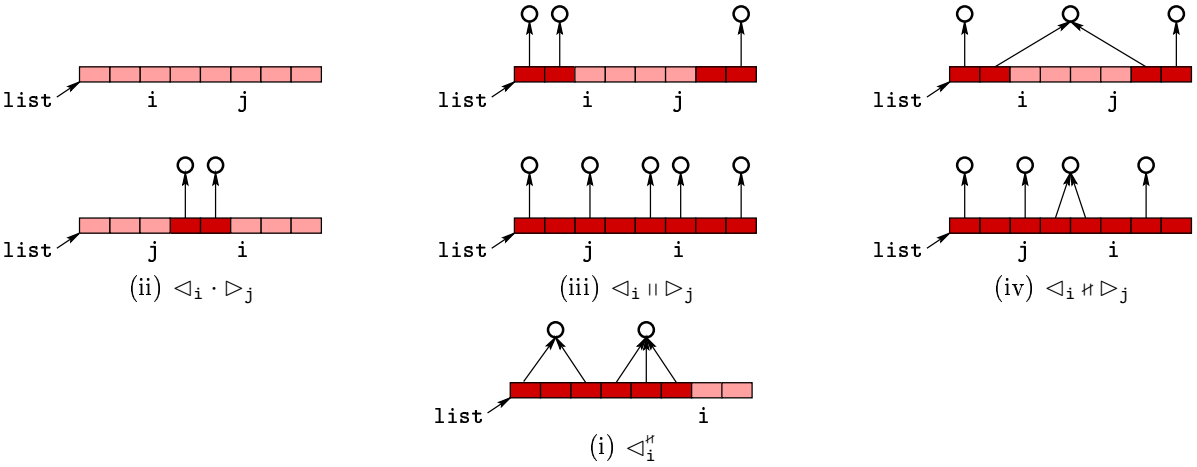
(i) $\diamond_{\mathtt{i}}$ (ii) $\triangleleft_{\mathtt{i}}$ (iii) $\triangleright_{\mathtt{i}}$

Figure 2: Connections described by primitive directions

$*$ describes any injective connection.

To abstract more properties of connections, we define *direction expressions*. A *direction expression* can be a primitive direction, or it can be built from the four operators specified below. In the following, we say a connection *satisfies* a direction if it can be described by the direction. Examples of direction expressions are given in Figure 3.

- $e \cdot e'$ describes any connection that satisfies both e and e'. Operator $\cdot$ is called the *meet* operator. When clear of the context, $\cdot$ is left out.
- $e \curlywedge e'$ describes any connection that can be split into two connections, such that one satisfies e and the other satisfies e'. Operator $\curlywedge$ is called the binary *mix* operator.
- $e \shortparallel e'$ describes any connection that can be split into two connections with *disjoint* target objects such that one satisfies e and the other satisfies e'. Operator $\shortparallel$ is called the *join* operator.
- $e^{\curlywedge}$ describes any connection whose domain satisfies e. Operator $\curlywedge$ is called the unary *mix* operator.



(ii) $\triangleleft_{\mathtt{i}} \cdot \triangleright_{\mathtt{j}}$ (iii) $\triangleleft_{\mathtt{i}} \shortparallel \triangleright_{\mathtt{j}}$ (iv) $\triangleleft_{\mathtt{i}} \curlywedge \triangleright_{\mathtt{j}}$

(i) $\triangleleft_{\mathtt{i}}^{\curlywedge}$

Figure 3: Examples of Direction Expressions

Finally, let Expr be the set of all direction expressions. We define an equivalence relation $\equiv$ on Expr: $e \equiv e'$ if and only if the same set of connections satisfy e and e'. An equivalence class for $\equiv$ is called a *direction*. The set of all directions, Dir, is thus the *quotient* of Expr by $\equiv$, i.e., $\mathsf{Expr}/\equiv$.

According to the above definitions, injectivity of connections can be inductively deduced from direction expressions. More precisely, a direction is a *comb* if it describes *injective* connections only. Combness of a direction d, written $\mathsf{Comb}(d)$, can be computed inductively from Table 1 (proof in Section B.1).

$\mathsf{Comb}(d \cdot d') = \mathsf{Comb}(d) \vee \mathsf{Comb}(d')$ $\neg\mathsf{Comb}(d \curlywedge d'), \quad (generalcase)$ $\neg\mathsf{Comb}(d^{\curlywedge}), \quad (generalcase)$	$\mathsf{Comb}(d \shortparallel d') = \mathsf{Comb}(d) \wedge \mathsf{Comb}(d')$

Table 1: Combness of direction expressions

Combness of directions improve the precision of the points-to analysis: from the injectivity of connections, one may deduce the *shape* of a container. On the other hand, we also need to tell whether an object is attached at a particular container position. Obtaining the latter information involves comparing the domains of two connections. Therefore, we define the following relations on directions:

- *Disjointness* tells whether two directions describe connections with disjoint domains;
- *Coincidence* tells whether two directions describe connections whose domains are *singletons* holding the *same* position.

Table 2 defines predicates Coincide and Disjoint which test these relations. Notice Disjoint is not the complement of Coincide. For instance, given $d = (\diamond_{\mathtt{i}} \lhd_{\mathtt{j}}) \between \diamond_{\mathtt{k}}$ and $d' = \diamond_{\mathtt{i}} \parallel (\diamond_{\mathtt{k}} \rhd_{\mathtt{j}})$, neither $\mathsf{Coincide}(d, d')$ nor $\mathsf{Disjoint}(d, d')$ hold.

$\mathsf{Coincide}(\diamond_{\mathtt{i}}, \diamond_{\mathtt{i}})$	$\mathsf{Disjoint}(\lhd_{\mathtt{i}}, \diamond_{\mathtt{i}}), \mathsf{Disjoint}(\diamond_{\mathtt{i}}, \rhd_{\mathtt{i}}), \mathsf{Disjoint}(\lhd_{\mathtt{i}}, \rhd_{\mathtt{i}})$
$\mathsf{Coincide}(d, d_1 \cdot d_2) = \mathsf{Coincide}(d, d_1) \vee \mathsf{Coincide}(d, d_2)$	$\mathsf{Disjoint}(d, d_1 \cdot d_2) = \mathsf{Disjoint}(d, d_1) \vee \mathsf{Disjoint}(d, d_2)$
$\mathsf{Coincide}(d, d_1 \parallel d_2) = \mathsf{Coincide}(d, d_1) \wedge \mathsf{Coincide}(d, d_2)$	$\mathsf{Disjoint}(d, d_1 \parallel d_2) = \mathsf{Disjoint}(d, d_1) \wedge \mathsf{Disjoint}(d, d_2)$
$\mathsf{Coincide}(d, d_1 \between d_2) = \mathsf{Coincide}(d, d_1) \wedge \mathsf{Coincide}(d, d_2)$	$\mathsf{Disjoint}(d, d_1 \between d_2) = \mathsf{Disjoint}(d, d_1) \wedge \mathsf{Disjoint}(d, d_2)$
$\mathsf{Coincide}(d, d_2^{\between}) = \mathsf{Coincide}(d, d_2)$	$\mathsf{Disjoint}(d, d_2^{\between}) = \mathsf{Disjoint}(d, d_2)$

Table 2: Coincidence and disjointness of directions

3.2 Abstract Interpretation for Directions

We use *abstract interpretation* [5] to enable systematic proofs about directions. Iter is a finite set of iterator variables. Formally, a connection is a pair of mappings (μ, ν), where $\mu : \mathbb{N} \to \mathbb{N}$ maps container positions to the objects attached at the positions, and $\nu : \mathsf{Iter} \to \mathbb{N}$ maps iterator variables to the positions they indicate.[3] Let Conn be the set of all connections.

- First, we define function $\gamma : \mathsf{Expr} \to 2^{\mathsf{Conn}}$ for direction expressions, as given in Table 3. Directions are equivalence classes of Expr for the following relation:

$$\forall e, e' \in \mathsf{Expr} : e \equiv e' \iff^{\text{def}} \gamma(e) = \gamma(e').$$

  From this relation, we can define the *concretizaton* function $\gamma : \mathsf{Dir} \to 2^{\mathsf{Conn}}$. Then, the link with the intuitive definition of directions is the following: a connection, c, satisfies a direction, d, if and only if $c \subseteq \gamma(d)$.

- We then define a *partial ordering* $\sqsubset$ over Dir:

$$d \sqsubset d' \iff^{\text{def}} \gamma(d) \subset \gamma(d').$$

  One may easily check the following orderings on directions:

$$d'' \sqsubseteq d \cdot d' \implies d'' \sqsubseteq d \wedge d'' \sqsubseteq d' \qquad d \parallel d' \sqsubseteq d'' \implies d \sqsubseteq d'' \wedge d' \sqsubseteq d''.$$

  As a result, Dir can be structured as a complete lattice ordered by $\sqsubset$. The *join* and *meet* operators of the lattice are $\parallel$ and $\cdot$, respectively. We prove the completeness of the lattice in Section B.4.

- Finally, the *abstraction* function $\alpha : 2^{\mathsf{Conn}} \to \mathsf{Dir}$ is built from γ. Given $C \subseteq \mathsf{Conn}$, α is defined as,

$$\alpha(C) = \min_{\sqsubset} \big\{ d \in \mathsf{Dir} : C \subseteq \gamma(d) \big\}.$$

  The minimum exists because the lattice of Dir is complete. Indeed, an equivalent definition for $\alpha(C)$ is the (infinite) meet of all directions that C satisfies.

$$
\begin{array}{rcl}
\gamma(*) & = & \{(\mu,\nu) \mid \mathsf{Injective}(\mu)\} \\
\gamma(\varnothing) & = & \{(\mu,\nu) \mid \mathsf{Domain}(\mu) = \varnothing\} \\
\gamma(\diamond_{\mathtt{i}}) & = & \{(\mu,\nu) \mid \mathsf{Domain}(\mu) = \{\nu(\mathtt{i})\}\} \\
\gamma(\triangleleft_{\mathtt{i}}) & = & \{(\mu,\nu) \mid (\forall p \in \mathsf{Domain}(\mu) : p < \nu(\mathtt{i})) \wedge \mathsf{Injective}(\mu)\} \\
\gamma(\triangleright_{\mathtt{i}}) & = & \{(\mu,\nu) \mid (\forall p \in \mathsf{Domain}(\mu) : p > \nu(\mathtt{i})) \wedge \mathsf{Injective}(\mu)\} \\
\gamma(e \cdot e') & = & \gamma(e) \cap \gamma(e') \\
\gamma(e \shuffle e') & = & \{(\mu,\nu) \mid \exists \mu_e, \mu_{e'} : (\mu_e,\nu) \in \gamma(e) \wedge (\mu_{e'},\nu) \in \gamma(e') \wedge \mathsf{Range}(\mu_e) \cap \mathsf{Range}(\mu_{e'}) = \varnothing \\
 & & \quad \wedge (\forall p \in \mathsf{Domain}(\mu) : \mu(p) = \mu_e(p) \vee \mu(p) = \mu_{e'}(p))\} \\
\gamma(e \barwedge e') & = & \{(\mu,\nu) \mid \exists \mu_e, \mu_{e'} : (\mu_e,\nu) \in \gamma(e) \wedge (\mu_{e'},\nu) \in \gamma(e') \\
 & & \quad \wedge (\forall p \in \mathsf{Domain}(\mu) : \mu(p) = \mu_e(p) \vee \mu(p) = \mu_{e'}(p))\} \\
\gamma(e^{\barwedge}) & = & \{(\mu,\nu) \mid \exists \mu_e : (\mu_e,\nu) \in \gamma(e) \wedge \mathsf{Domain}(\mu) = \mathsf{Domain}(\mu_e)\}
\end{array}
$$

Table 3: Concretization functions for direction expressions

To ensure α and γ preserve the relative precision over sets of connections and directions. We prove that α and γ are monotonic over $\sqsubset$:

$$\forall C \subseteq \mathsf{Conn} : C \subseteq C' \implies \alpha(C) \sqsubseteq \alpha(C') \qquad \forall d \in \mathsf{Dir} : d \sqsubseteq d' \implies \gamma(d) \subseteq \gamma(d') \tag{1}$$

In addition, the following equation says that γ is a safe approximation and that γ is the inverse image of α, ensuring that no information is lost during concretization:

$$\forall C \subseteq \mathsf{Conn} : C \subseteq \gamma(\alpha(C)) \qquad \forall d \in \mathsf{Dir} : d = \alpha(\gamma(d)) \tag{2}$$

Proof of (2) can be found in Section B.5. The following result summarizes these properties in the unified model of Cousot and Cousot [5] for program analysis.

Theorem 1 *Abstraction and concretization functions α and γ define a* Galois connection *from the lattice of sets of connections to the lattice of directions.*

We use this result in the appendix to prove properties of the direction abstraction (e.g., algebraic properties of operators), and to prove the analysis safety (i.e., it computes a correct approximation for each container layout).

4 Points-to Graphs

4.1 The Node Naming Scheme

In points-to graphs, heap objects are represented by *heap nodes* named as n^H_X: X, the *reference set*, holds variables that are *simultaneously* pointing to the objects at this moment; and H, the *history set*, records all variables that *ever pointed* to the objects at some past execution. Among heap nodes, we further distinguish between *object nodes* and *summary nodes*: an object node abstracts a *single object* and has a *non-empty* reference set; a summary node abstracts a *pool of objects* and has an *empty* reference set. Object and summary nodes are distinguished by predicates Obj and Sum, respectively. Intuitively, object nodes keep precise information about single objects, thus, enable "killing" on assignments to object fields. Summary nodes, on the other hand, provide a mechanism to group objects together, which is necessary to make the graph finite. Additional remarks about this naming scheme are,

- The node naming scheme using reference sets was first introduced by [18]. This is used to determine the *compatibility* of heap nodes: two nodes are *compatible* if they may *simultaneously* abstract (non-empty sets of) heap object. Since any variable may not simultaneously point to multiple heap objects, nodes n^H_X and $n^{H'}_{X'}$ with *different* but *overlapping* reference sets (i.e., $X \neq X' \wedge X \cap X' \neq \varnothing$) can *not*

[3] To simplify the formalism, heap objects and container positions are labeled by integers.

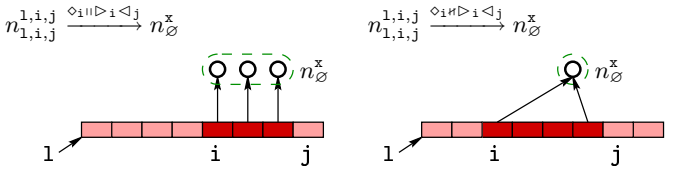


Figure 4: Container edges and direction labels

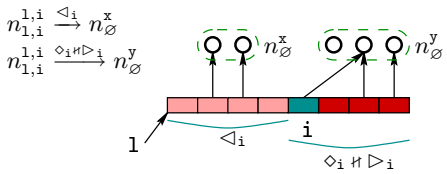


Figure 5: Multiple edges

simultaneously abstract heap objects. In other words, they are not compatible. In a similar way, n^H_X and $n^{H'}_{X'}$ with $X = X' \neq \varnothing$ and $H \neq H'$ are *not* compatible either. Node compatibility can improve the analysis' precision because edges that connect incompatible nodes are spurious and can be removed from the graph. We introduce predicate Compat to compute such infromation:

$$\mathsf{Compat}(n^H_X, n^{H'}_{X'}) \iff^{\text{def}} (X = X' \wedge H = H') \vee X \cap X' = \varnothing.$$

- History sets make our node naming scheme different from the original one proposed in [19] and further extentions proposed in [18, 20, 4]. Indeed, history sets are used to create more object/summary nodes, preserving useful context information. This is similar to other heuristics [18, 20, 4], which resorts to allocation sites or data types to achieve the same goal.
- The points-to graph describes a *partition of memory*. In fact, for any given runtime program point, a heap object can only be abstracted by *one* heap node in the graph. However, the node naming scheme is *store-less*, meaning that nodes in a points-to graph do *not* describe a *fixed* partition of memory/store. In other words, whether an object is abstracted by a node or another depends on the program point. This store-less approach is critical for the precise handling of random container updates.

4.2 Definition of Points-to Graphs

We define a points-to graph $G = (V, N, R, E, C)$ as follows.

- V is the set of *program variables*.
- N is the set of *heap nodes*.
- $R \subseteq V \times \{o \in N \mid \mathsf{Obj}(o)\}$ is the set of *variable edges*. It holds directed edges of the form $\mathtt{v} \longrightarrow o$, indicating that variable $\mathtt{v}$ *must* point to the *single* object abstracted by *object* node o.
- $E \subseteq N \times \mathsf{Fld} \times N$ is the set of *field edges*. It holds labeled directed edges of the form, $o \xrightarrow{\mathtt{f}} o'$, indicating that *field* $\mathtt{f}$ of an object abstracted by o *may point to* an object abstracted by o'.
- $C \subseteq N \times \mathsf{Dir} \times N$ is the set of *container edges*. It holds labeled directed edges of the form, $l \xrightarrow{d} o$, indicating that *every* connection between a container abstracted by l and an object abstracted by o satisfies d.

Note that, edges in R and E represent physical pointers, whereas C only abstracts logical connections between containers and their elements. Figure 4 and 5 give examples of container layouts and the corresponding container edges.[4]

To re-address the node compatibility and the partition property, Figure 6 gives two examples where $\mathtt{x}$ and $\mathtt{y}$ may point to the same location or not. The two heap layouts satisfy the partition property: either node $n^{\mathtt{x},\mathtt{y}}_{\mathtt{x},\mathtt{y}}$ abstracts an object simultaneously referenced by $\mathtt{x}$ and $\mathtt{y}$ or nodes $n^{\mathtt{x},\mathtt{y}}_{\mathtt{x}}$ and $n^{\mathtt{y}}_{\mathtt{y}}$ abstract distinct objects.

To ensure the graph is finite, two important steps are applied after each points-to graph update:

- A garbage-collection scheme removes any node (along with its adjacent edges) when it is not reachable from any variable node. Container edges labeled $\varnothing$ are removed as well.

[4] A container node can have multiple outgoing edges (targeting distinct nodes). Notice an object abstracted by $n^{\mathtt{y}}_{\varnothing}$ is not required to be an element of the container.

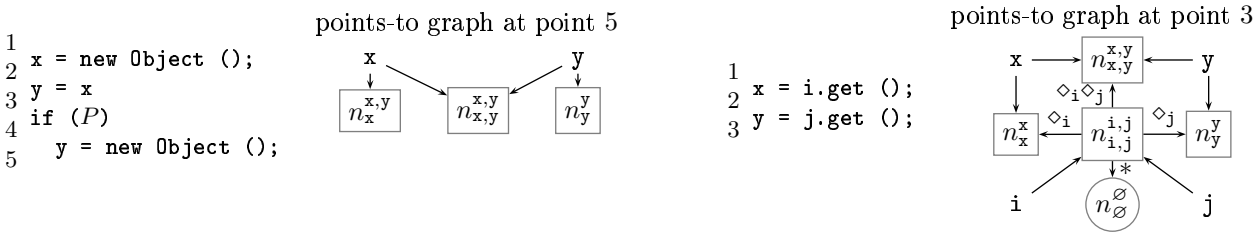


Figure 6: Node compatibility

- To ensure that a finite set of iterator variables generate a *finite number* of directions, labels are simplified according to equivalence of direction expressions. The formal rules are given in the Appendix. A special case is that directions of the form $e \curlywedge e$ are replaced by $e^{\curlywedge}$. This step can be seen as a "widening" transformation, which conservatively drops injective properties when they are unlikely to bring any precision improvement. Local safety proofs in Section B.6.

5 Points-to Analysis

The points-to analysis to be presented is intra-procedural except for the dedicated interpretation of container-related methods.

5.1 An Intuitive Flavor of the Analysis

Let us first illustrate the points-to analysis in the following example that reverses a list. We would like to infer that the `reverse()` method preserves combness.

```
1 static void reverse (Container c, Container d) {
2   Iterator i = c.last ();
3   Iterator j = d.first ();
4   while (!i.atBegin ()) {
5     Object o = i.get ();
6     j.put (o);
7     j.advance ();
8     i.retreat ();
9   }
}
```

Figure 7 describes the first two iterations of the `while` loop. For each run-time program point, it gives a concrete heap layout in parallel with the points-to graph corresponding to the layout. Initially, we assume that `c` and `d` point to distinct containers, that `c` is a comb, and that `d` is empty. In the beginning, the graph holds three nodes: object nodes $n^{\mathtt{c}}_{\mathtt{c}}$ and $n^{\mathtt{d}}_{\mathtt{d}}$ which abstract container objects referenced by `c` and `d`, and a summary node, $n^{\varnothing}_{\varnothing}$, which abstracts elements of $n^{\mathtt{c}}_{\mathtt{c}}$. The container edge between $n^{\mathtt{c}}_{\mathtt{c}}$ and $n^{\varnothing}_{\varnothing}$ is labeled $*$, describing an injective connection.

Point 4_1. The `while` loop is entered for the first time. Iterator `i` (resp. `j`) is attached to the container referenced by c (resp. d). These container nodes are renamed to $n^{\mathtt{c},\mathtt{i}}_{\mathtt{c},\mathtt{i}}$ and $n^{\mathtt{d},\mathtt{j}}_{\mathtt{d},\mathtt{j}}$ to reflect the attachments of the iterators.

Point 5_1, `o = i.get()`. The object attached at the position of `i` is now assigned to variable `o`. Since iterator `i` is associated with $n^{\mathtt{c},\mathtt{i}}_{\mathtt{c},\mathtt{i}}$, this object must be abstracted by $n^{\varnothing}_{\varnothing}$ (elements of $n^{\mathtt{c},\mathtt{i}}_{\mathtt{c},\mathtt{i}}$). Then, the object is "materialized" [18] from summary node $n^{\varnothing}_{\varnothing}$ into a new object node. This new object node, now pointed to by `o`, is named $n^{\mathtt{o}}_{\mathtt{o}}$. The connection from $n^{\mathtt{c},\mathtt{i}}_{\mathtt{c},\mathtt{i}}$ to $n^{\mathtt{o}}_{\mathtt{o}}$ is captured by $\diamond_{\mathtt{i}}$.

Point 6_1, `j.put(o)`. Since iterator `j` is assocaited with container node $n^{\mathtt{d},\mathtt{j}}_{\mathtt{d},\mathtt{j}}$, $n^{\mathtt{o}}_{\mathtt{o}}$ is connected to $n^{\mathtt{d},\mathtt{j}}_{\mathtt{d},\mathtt{j}}$ by a container edge labeled $\diamond_{\mathtt{j}}$.

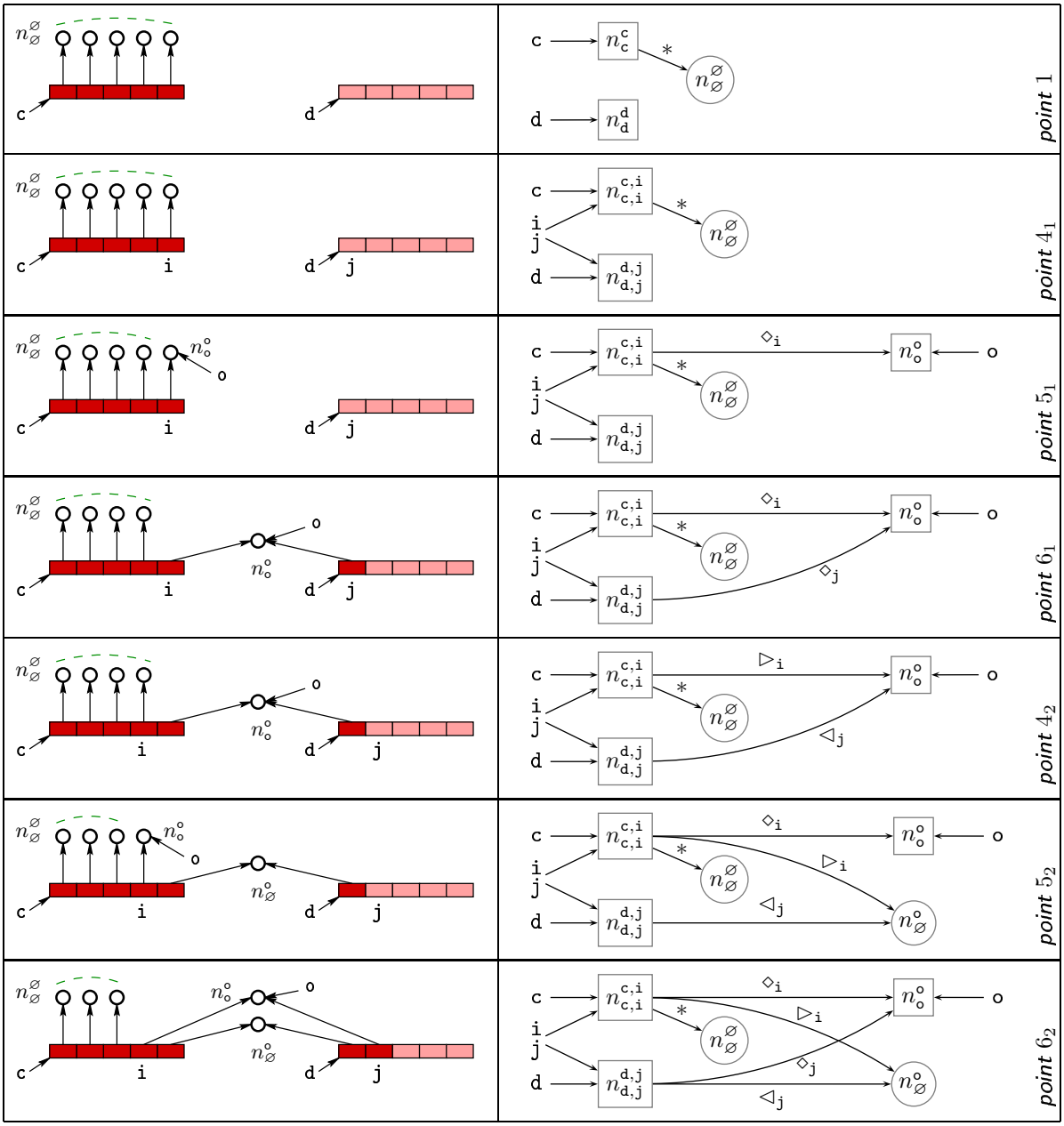


Figure 7: Points-to analysis for `reverse`

Point 4_2**.** Iterator `i` (resp. `j`) is moved backward (resp. forward) before entering the next iteration. Accordingly, the direction labels are updated: $\diamond_{\mathtt{i}}$ is replaced by $\triangleright_{\mathtt{i}}$, $\diamond_{\mathtt{j}}$ is replaced by $\triangleleft_{\mathtt{j}}$.

Point 5_2**,** `o = i.get()`**.** First, we "kill" the outgoing edges of o, i.e., removing the edge from `o` to $n^{\circ}_{\mathtt{o}}$. This also involves removing `o` from the reference set of $n^{\circ}_{\mathtt{o}}$. As a result, $n^{\circ}_{\mathtt{o}}$ becomes a summary node $n^{\circ}_{\varnothing}$. We call such a process *node summarization*. Then, the object attached at the position of `i` needs to be extracted from the elements of $n^{\mathtt{c,i}}_{\mathtt{c,i}}$. Note that, both $n_{\varnothing}$ and $n^{\circ}_{\varnothing}$ are attached at $n^{\mathtt{c,i}}_{\mathtt{c,i}}$. However, since the edge targetting $n^{\circ}_{\varnothing}$ is labeled as $\triangleright_{\mathtt{i}}$, which is disjoint with $\diamond_{\mathtt{i}}$, the object can not be abstracted by $n^{\circ}_{\varnothing}$. Therefore, we only materizalize from $n^{\varnothing}_{\varnothing}$. Again, the new object node is named $n^{\circ}_{\mathtt{o}}$.

Point 6_2, `j.put(o)`. the same as **Point** 6_1. Note that, the outgoing edges of $n^{\mathtt{d,j}}_{\mathtt{d,j}}$ and $n^{\mathtt{c,i}}_{\mathtt{c,i}}$ are labeled by injective directions, meaning that both containers are comb.

Further iterations follow a similar pattern. A fixed point is actually reached at the third iteration. To conclude this example, we would like to emphasize three key properties of the analysis that make us successfully detect the preservation of combness.

- Node materialization helps to restore precise points-to information when an object abstracted by a summary node is assigned to a variable (e.g., `o = i.get()`).
- Because of history sets, after the "kill" of variable `o`, $n^{\circ}_{\mathtt{o}}$—which abstracts the "tail" elements of $n^{\mathtt{c,i}}_{\mathtt{c,i}}$—is summarized into $n^{\circ}_{\varnothing}$, avoiding spurious confusion with "head" elements abstracted by $n^{\varnothing}_{\varnothing}$.
- Comparing $\diamond_{\mathtt{i}}$ and $\triangleright_{\mathtt{i}}$ through $\mathsf{Disjoint}(\diamond_{\mathtt{i}}, \triangleright_{\mathtt{i}})$ avoids materialization from $n^{\circ}_{\varnothing}$.

5.2 Strong Nullification of Variables

A strong nullification, for a points-to graph, means the "kill" (i.e., removal) of points-to edges. More specifically, a strong nullification of variables/fields/containers correspond to the removal of variable/field/container edges, respectively. The ability to "kill" is key to the high precision of this analysis. We define an auxiliary "kill" function to implement the strong nullification of *variables*. The "kill" functions for fields or container positions are implemented directly into transfer functions.

Given a points-to graph, $G = (V, N, R, E, C)$, $\mathsf{KillVariable}_v(G)$ implements the strong nullification of `v`, i.e., `v = null`. The function first removes all outgoing edges of `v`. Then, `v` is removed from the reference set of each node, i.e., n^H_X is renamed to $n^H_{X\setminus\{\mathtt{v}\}}$. If node $n^H_{X\setminus\{\mathtt{v}\}}$ already exists in the graph, edges of the two nodes need to be merged to ensure *at most* one edge between any two nodes in the graph.

- When both have an incoming container edge from the *same* container node, merging of the two edges shall preserve combness. We define operator $\cup^{\shortparallel}$ to implement such a merge:

$$C \cup^{\shortparallel} C' \stackrel{\text{def}}{=} (C \setminus C') \cup (C' \setminus C) \cup \left\{o \xrightarrow{d \shortparallel d'} o' \mid o \xrightarrow{d} o' \in C \wedge o \xrightarrow{d'} o' \in C'\right\}.$$

- When both merging nodes have an outgoing container edge to the *same* node, combness is not preserved because a single element may be attached in multiple containers. Operator $\cup^{\curlywedge}$ implements this merge:

$$C \cup^{\curlywedge} C' \stackrel{\text{def}}{=} (C \setminus C') \cup (C' \setminus C) \cup \left\{o \xrightarrow{d \curlywedge d'} o' \mid o \xrightarrow{d} o' \in C \wedge o \xrightarrow{d'} o' \in C'\right\}.$$

In the points-to analysis, KillVariable is the *only way* to generate a summary node. $\mathsf{KillVariable}_v(G)$ is defined below.[5] Examples are given in Figure 8.

$$\begin{array}{l}
\mathsf{KillVariable}_{\mathtt{v}}(V, N, R, E, C) = (V, N', R', E', C') \\
\quad (N', R', E', C') \leftarrow (N, R, E, C) \\
\quad n^H_X \in N \wedge \mathtt{v} \in X : N' \leftarrow N' \setminus \{n^H_X\} \cup \{n^H_{X\setminus\{\mathtt{v}\}}\}; R' \leftarrow R' \setminus \{\mathtt{v} \rightarrow n^H_X\} \\
\quad o \xrightarrow{\mathtt{f}} n^H_X \in E \wedge \mathtt{v} \in X : E' \leftarrow E' \setminus \{o \xrightarrow{\mathtt{f}} n^H_X\} \cup \{o \xrightarrow{\mathtt{f}} n^H_{X\setminus\{\mathtt{v}\}}\} \\
\quad n^H_X \xrightarrow{\mathtt{f}} o \in E \wedge \mathtt{v} \in X : E' \leftarrow E' \setminus \{n^H_X \xrightarrow{\mathtt{f}} o\} \cup \{n^H_{X\setminus\{\mathtt{v}\}} \xrightarrow{\mathtt{f}} o\} \\
\quad o \xrightarrow{d} n^H_X \in C \wedge \mathtt{v} \in X : C' \leftarrow C' \setminus \{o \xrightarrow{d} n^H_X\} \cup^{\shortparallel} \{o \xrightarrow{d} n^H_{X\setminus\{\mathtt{v}\}}\} \\
\quad n^H_X \xrightarrow{d} o \in C \wedge \mathtt{v} \in X \wedge \mathsf{Compat}(n^H_X, n^H_{X\setminus\{\mathtt{v}\}}) : C' \leftarrow C' \setminus \{n^H_X \xrightarrow{d} o\} \cup^{\curlywedge} \{n^H_{X\setminus\{\mathtt{v}\}} \xrightarrow{d} o\} \\
\quad n^H_X \xrightarrow{d} o \in C \wedge \mathtt{v} \in X \wedge \neg\mathsf{Compat}(n^H_X, n^H_{X\setminus\{\mathtt{v}\}}) : C' \leftarrow C' \setminus \{n^H_X \xrightarrow{d} o\} \cup^{\shortparallel} \{n^H_{X\setminus\{\mathtt{v}\}} \xrightarrow{d} o\}.
\end{array}$$

[5] For this function and graph transformers to come, rules of the form $C : A$ use a side-effect notation where C is a "for-all" condition which "triggers" action A. Rules apply in-order but conditions are exclusive when present.

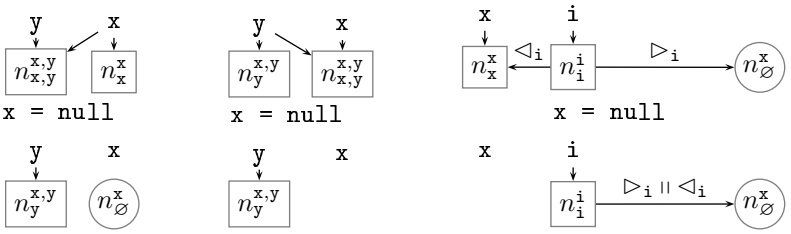


Figure 8: Strong nullifications of variables

Figure 9: Materialization

5.3 Node Materialization

For best precision, an object abstracted by a summary node needs to be "materialized" to an object node when it is assigned to a variable [18]. Figure 9 gives an example of such a process. As a result, in a points-to graph, variables can not directly point to summary nodes. We define an auxiliary function to implement such a graph transformation: $\mathsf{Materialize}_{\mathtt{v},n^H_\varnothing}(G)$ "extracts" an object node from a summary node $n^H_\varnothing$ and connects it with variable `v`. Basically, the function duplicates $n^H_\varnothing$ (including its adjacent edges) with a name $n^{H\cup\{\mathtt{v}\}}_{\mathtt{v}}$ and adds $v \longrightarrow n^{H\cup\{\mathtt{v}\}}_{\mathtt{v}}$ to the graph.

We also materialize nodes through container edges. When materializing an object node for `i.get()`, one needs a new container edge to record the fact that the materialized node is attached at $\diamond_{\mathtt{i}}$. We thus generalize the concept of materialization to "transfer" an object from an *object or summary* node to an object node. $\mathsf{Materialize}_{\mathtt{v},n^H_X}(G)$ duplicates n^H_X (including its adjacent edges) with a name $n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}}$ and links `v` to the duplicated node.[6] Materialize is formally defined below.

$$\begin{aligned}\mathsf{Materialize}_{\mathtt{v},n^H_X}(V,N,R,E,C) = \big(&V, N \cup \{n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}}\}, R \cup \{\mathtt{v} \longrightarrow n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}}\} \cup \{\mathtt{w} \longrightarrow n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}} \mid \mathtt{w} \longrightarrow n^H_X \in R\},\\ &E \cup \{o \xrightarrow{\mathtt{f}} n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}} \mid o \xrightarrow{\mathtt{f}} n^H_X \in E\} \cup \{n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}} \xrightarrow{\mathtt{f}} o \mid n^H_X \xrightarrow{\mathtt{f}} o \in E\},\\ &C \cup \{o \xrightarrow{d} n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}} \mid o \xrightarrow{d} n^H_X \in C\} \cup \{n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}} \xrightarrow{d} o \mid n^H_X \xrightarrow{d} o \in C\}\big).\end{aligned}$$

5.4 Transfer Functions

In the following, $X[b/a]$ denotes the substitution of a with b in each element of set X.

5.4.1 The Classical Part

This part discusses transfer functions for pointer assignments that are not specific to containers. We use an auxiliary function $\mathsf{Link}_{\mathtt{v},n^H_X}(G)$ to connect a variable `v` to an *object* node n^H_X ($X \neq \varnothing$) and then to rename n^H_X to $n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}}$. Link assumes that a strong nullification of variable `v` is applied before, therefore $n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}}$ does not name-conflict with any other node. Here is the formal definition of Link:

$$\mathsf{Link}_{\mathtt{v},n^H_X}(V,N,R,E,C) = \big(V, N[n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}}/n^H_X], R \cup \{\mathtt{v} \longrightarrow n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}}\}, E[n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}}/n^H_X], C[n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}}/n^H_X]\big).$$

Transfer functions are specified in Figure 10.

`v = w`, `v = null`, and `v = new`. It involves a strong nullification of `v`, followed by linking `v` to an object node.

`v.f = w`. It implements a strong nullification of object field `v.f`: it removes every outgoing field-edge of *object* nodes pointed by `v` that are labeled by `f`. As opposed to KillVariable, this involves no renaming, thus no merging of nodes is needed.

[6] Safety of materialization from an object node (when X is not empty) is similar to safety of joining points-to graphs at control-flow merges in Sagiv, Reps and Wilhelm technique [18]. Indeed, nodes n^H_X and $n^{H\cup\{\mathtt{v}\}}_{X\cup\{\mathtt{v}\}}$ may not simultaneously abstract a concrete object since they are not *compatible*.

`v = null`	$G \leftarrow \mathsf{KillVariable}_{\mathtt{v}}(G)$
`v = new Object()`	$G \leftarrow \mathsf{KillVariable}_{\mathtt{v}}(G)$ $N \leftarrow N \cup \{n^{\mathtt{v}}_{\mathtt{v}}\}; R \leftarrow R \cup \{\mathtt{v} \rightarrow n^{\mathtt{v}}_{\mathtt{v}}\}$
`v = w`	$G \leftarrow \mathsf{KillVariable}_{\mathtt{v}}(G)$ $\mathtt{w} \rightarrow n^H_X \in R : G \leftarrow \mathsf{Link}_{\mathtt{v},n^H_X}(G)$
`v = w.f`	$G \leftarrow \mathsf{KillVariable}_{\mathtt{v}}(G)$ $\mathtt{w} \rightarrow o \in R \wedge o \xrightarrow{\mathtt{f}} n^H_X \in E \wedge \mathsf{Obj}(n^H_X) : G \leftarrow \mathsf{Link}_{\mathtt{v},n^H_X}(G)$ $\mathtt{w} \rightarrow o \in R \wedge o \xrightarrow{\mathtt{f}} n^H_\varnothing \in E : G \leftarrow \mathsf{Materialize}_{\mathtt{v},n^H_\varnothing}(G)$
`v.f = w`	$\mathtt{v} \rightarrow o \in R \wedge o \xrightarrow{\mathtt{f}} o' \in E : E \leftarrow E \setminus \{o \xrightarrow{\mathtt{f}} o'\}$ $\mathtt{v} \rightarrow o \in R \wedge \mathtt{w} \rightarrow o' \in R \wedge \mathsf{Compat}(o, o') : E \leftarrow E \cup \{o \xrightarrow{\mathtt{f}} o'\}$

Figure 10: Transfer functions of non-container specific assignments

`i = l.first()` `i = l.last()`	$G \leftarrow \mathsf{KillVariable}_{\mathtt{i}}(G)$ $C \leftarrow C[*/\diamond_{\mathtt{i}}, */\triangleleft_{\mathtt{i}}, */\triangleright_{\mathtt{i}}]$ $\mathtt{l} \rightarrow n^H_X \in R : G \leftarrow \mathsf{Link}_{\mathtt{i},n^H_X}(G)$
`i.advance()`	$C \leftarrow C[\triangleleft_{\mathtt{i}}/\diamond_{\mathtt{i}}, */\triangleright_{\mathtt{i}}]$
`i.retreat()`	$C \leftarrow C[\triangleright_{\mathtt{i}}/\diamond_{\mathtt{i}}, */\triangleleft_{\mathtt{i}}]$

Figure 11: Transfer functions of iterator operations

`v = w.f`. It starts with a strong nullification of `v`. Then, if `w.f` points to an object node, `v` is directly linked to that object node. On the other hand, if `w.f` points to a summary node, an object node is first materialized, then it is linked to `v`.

5.4.2 Iterator Operations

We only consider iterator variables that are *unaliased*. This simplifying assumption has two consequences:

- Calling a method on an iterator variable will not affect iterator objects referenced by other variables.
- The attachment of an iterator to a container node can be represented as a *variable edge* with the iterator variable appearing in the reference and history sets of the container node. Such variable edges capture *must-attachment* properties,[7] which is key to the strong nullification of container fields.

Figure 11 gives transfer functions for iterator operations that involve no container update/access. The explanation is given below:

`i=l.first()` **and** `i=l.last()`. It involves a strong nullification of iterator `i` and linking `i` to container l. In addition, since `i` is re-attached, previous direction labels involving `i` become obsolete, thus, replaced by $*$.[8]

`i.advance()`. Since `i` moves forward, previous direction labels involving `i` are updated: $\diamond_{\mathtt{i}}$ is replaced by $\triangleleft_{\mathtt{i}}$ and $\triangleright_{\mathtt{i}}$ by $*$.

`i.retreat()`. Since `i` moves backward, the previous direction labels involving `i` are updated: $\diamond_{\mathtt{i}}$ is replaced by $\triangleright_{\mathtt{i}}$ and $\triangleleft_{\mathtt{i}}$ by $*$.

[7] Only *may-point-to* properties would be available if the attachment was indirect, i.e., if we connected an iterator *object node* to the container.

[8] As a consequence, a direction label of a container edge may only involve iterators currently attached to the container. Moreover, any outgoing container edge from summary container node has direction label $*$ or $*^{\prime\prime}$.

5.4.3 Container Operations

This section discusses iterator operations that update or access container elements. Different semantics have been proposed for such operators. We choose one that is both realistic and easy to adapt to a particular library: none of the iterator operations implicitly move iterators. Specifically, `i.delete()` removes the object at the position of `i` leaving `i` "between" two legal positions;[9] `i.insert(v)` adds an object *after* the position of `i`; `i.put(v)` and `i.get()` occur at the position of `i`.

`i.delete()`	$\mathtt{i} \rightarrow l \in R \wedge l \xrightarrow{d} o \in C : C \leftarrow C \setminus \{l \xrightarrow{d} o\} \cup \{l \xrightarrow{d \cdot (\lhd_\mathtt{i} \shortparallel \rhd_\mathtt{i})^\kappa} o\}$
`i.insert(v)`	$\mathtt{i} \rightarrow l \in R \wedge \mathtt{v} \rightarrow o \in R \wedge \mathsf{Compat}(l, o) : C \leftarrow C \cup^\kappa \{l \xrightarrow{\rhd_\mathtt{i}} o\}$
`i.put(v)`	$\mathtt{i} \rightarrow l \in R \wedge \mathtt{v} \rightarrow o \in R \wedge l \xrightarrow{d} o' \in C \wedge o \neq o' : C \leftarrow C \setminus \{l \xrightarrow{d} o'\} \cup \{l \xrightarrow{d \cdot (\lhd_\mathtt{i} \shortparallel \rhd_\mathtt{i})^\kappa} o'\}$ $\mathtt{i} \rightarrow l \in R \wedge \mathtt{v} \rightarrow o \in R \wedge \neg(l \xrightarrow{d} o \in C) \wedge \mathsf{Compat}(l, o) : C \leftarrow C \cup \{l \xrightarrow{\diamond_\mathtt{i}} o\}$ $\mathtt{i} \rightarrow l \in R \wedge \mathtt{v} \rightarrow o \in R \wedge l \xrightarrow{d} o \in C \wedge \neg(\diamond_\mathtt{i} \in d) : C \leftarrow C \setminus \{l \xrightarrow{d} o\} \cup \{l \xrightarrow{d \kappa \diamond_\mathtt{i}} o\}$ $\mathtt{i} \rightarrow l \in R \wedge \mathtt{v} \rightarrow o \in R \wedge l \xrightarrow{d} o \in C \wedge \diamond_\mathtt{i} \in d : C \leftarrow C \setminus \{l \xrightarrow{d} o\} \cup \{l \xrightarrow{d \kappa (d \cdot \diamond_\mathtt{i})} o\}$
`v = i.get()`	$G \leftarrow \mathsf{KillVariable}_\mathtt{v}(G)$ $\mathtt{i} \rightarrow l \in R \wedge l \xrightarrow{d} n_X^H \in C \wedge \mathsf{Obj}(n_X^H) \wedge \mathsf{Coincide}(\diamond_\mathtt{i}, d) : G \leftarrow \mathsf{Link}_{\mathtt{v}, n_X^H}(G)$ $\mathtt{i} \rightarrow l \in R \wedge l \xrightarrow{d} n_X^H \in C \wedge \neg\big(\mathsf{Obj}(n_X^H) \wedge \mathsf{Coincide}(\diamond_\mathtt{i}, d)\big) \wedge \neg\mathsf{Disjoint}(\diamond_\mathtt{i}, d) \wedge \mathsf{Comb}(d) :$ $G \leftarrow \mathsf{Materialize}_{\mathtt{v}, n_X^H}(G); C \leftarrow C \setminus \big\{l \xrightarrow{d} n_{X \cup \{\mathtt{v}\}}^{H \cup \{\mathtt{v}\}}\big\} \cup \big\{l \xrightarrow{\diamond_\mathtt{i} \cdot d} n_{X \cup \{\mathtt{v}\}}^{H \cup \{\mathtt{v}\}}\big\}$ $\mathtt{i} \rightarrow l \in R \wedge l \xrightarrow{d} n_X^H \in C \wedge \neg\big(\mathsf{Obj}(n_X^H) \wedge \mathsf{Coincide}(\diamond_\mathtt{i}, d)\big) \wedge \neg\mathsf{Disjoint}(\diamond_\mathtt{i}, d) \wedge \neg\mathsf{Comb}(d) :$ $G \leftarrow \mathsf{Materialize}_{\mathtt{v}, n_X^H}(G)$

Figure 12: Transfer functions of container operations

Transfer functions are given in Figure 12. Some examples are shown in Figure 13.

`i.delete()`. This is the container counterpart of `v.f = null`. The transfer function involves a strong nullification at the position of `i`. In other words, it removes any connection (container edge) whose domain can be described by $\diamond_\mathtt{i}$. Having the property,

$$\diamond_\mathtt{i} \cdot (\lhd_\mathtt{i} \shortparallel \rhd_\mathtt{i})^\kappa \equiv (\diamond_\mathtt{i} \cdot \lhd_\mathtt{i})^\kappa \shortparallel (\diamond_\mathtt{i} \cdot \rhd_\mathtt{i})^\kappa \equiv \varnothing$$

this step is implemented by a substitution of any label d by $d \cdot (\lhd_\mathtt{i} \shortparallel \rhd_\mathtt{i})^\kappa$. The actual edge removal, however, occurs at the garbage collection step when the direction label of an edge is equivalent to $\varnothing$.

`i.insert(v)`. An object is attached immediately follows the position of `i`. The transfer function adds a container edge labeled by $\rhd_\mathtt{i}$. Operator $\cup^\kappa$ is required, in this case, because the new container edge may conflict with an existing one.

`i.put(v)`. This operation combines the effect of a deletion and an insertion. Let l denote any container node pointed by `i` and let o denotes any node pointed by `v`. The transfer function starts with a strong nullification at the position of `i`: all edges targetting object nodes which are *not* pointed by `v` are updated the same way as `i.delete()`.

Then, we use the notation $\diamond_\mathtt{i} \in d$ to indicate that $\diamond_\mathtt{i}$ appears in *every expression* of d.[10] We use an important property of the analysis.

Lemma 1 *Let d be a direction such that $\diamond_\mathtt{i} \in d$, i.e., $\diamond_\mathtt{i}$ appears in every expression of d. For any configuration of iterator positions, d describes a connection whose domain is the current position of `i`. In formal terms, for any mapping ν, there is a mapping $\mu_\mathtt{i}$ such that $\mu_{\mathtt{i},\nu} \in \gamma(d)$ and $\mathsf{Domain}(\mu_\mathtt{i}) = \{\mathtt{i}\}$.*

[9] This is similar to the semantics of `remove()` in Java SDK

[10] This is equivalent to checking for $\diamond_\mathtt{i}$ in a flattened form expression of d, after exclusive primitive directions have been removed.

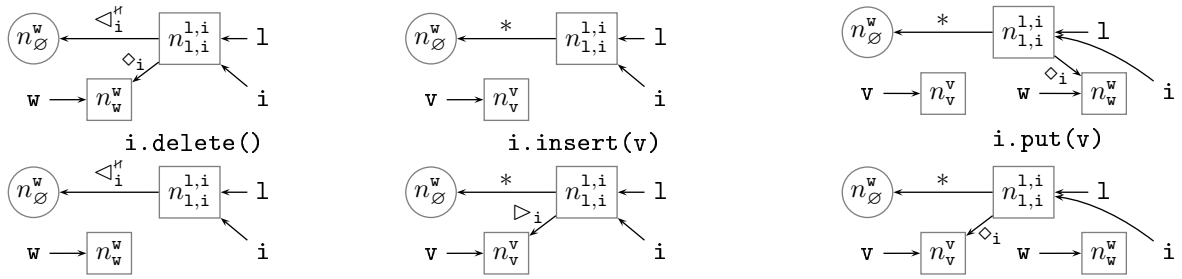


Figure 13: Container updates

There are three exclusive cases.

1. If there is no container edge targetting o from l, the transfer function simply adds $l \xrightarrow{\diamond_{\mathtt{i}}} o$ to the graph.
2. When there exists an edge $l \xrightarrow{d} o$ such that $\diamond_{\mathtt{i}} \in d$, the merged edge is labeled $d \curlywedge (d \cdot \diamond_{\mathtt{i}})$. Indeed, Lemma 1 enforces that a connection cannot satisfy $\diamond_{\mathtt{i}}$ without satisfying d as well. Therefore, the new connection from the position of `i` to the object abstracted by o is already described by d.
3. When there exists an edge $l \xrightarrow{d} o$ such that $\diamond_{\mathtt{i}} \notin d$, $l \xrightarrow{\diamond_{\mathtt{i}}} o$ is simply merged with the edge, and label d is substituted by $d \curlywedge \diamond_{\mathtt{i}}$. Notice direction $d \curlywedge \diamond_{\mathtt{i}}$ would be over-conservative when $\diamond_{\mathtt{i}} \in d$: it looses the information that the new connection satisfies *both* $\diamond_{\mathtt{i}}$ and d.

`v = i.get()`. This is the container counterpart of `v = w.f`. The transfer function starts with a strong nullification of variable `v`, followed by three exclusive cases.

1. If a node is linked to the container at the current position of `i` (i.e., $\mathsf{Coincide}(d, \diamond_{\mathtt{i}})$), the node is linked to `v`. See examples in Figure 14.a and 15.a.
2. If a node is connected to the container through a *comb* direction that *may describe* connections satisfying $\diamond_{\mathtt{i}}$ (i.e.,$\neg\mathsf{Disjoint}(d, \diamond_{\mathtt{i}})$), we *materialize* a new object node and link `v` to it. Compared to `v = w.f`, this materialization is not limited to summary nodes. Figure 14.c gives an example of materialization from object nodes. In addition, the container edge from the container to the materialized node is labeled $\diamond_{\mathtt{i}} \cdot d$.[11] See examples in Figure 14.b and 14.c.
3. This is similar to the previous case except that the container direction is *not* a comb. Materializing through a non-comb edge may create a node abstracting an object attached at several container positions: this forbids adding $\diamond_{\mathtt{i}}$ to the direction label. See examples in Figure 15.b and 15.c.

Note that, there must exist at least one container edge whose positions include $\diamond_{\mathtt{i}}$. This means that when none of the above rules apply, a programming error is detected (getting an element from an unattached iterator).

5.5 Merging Edges

Eventually, we structure points-to graphs as a join semi-lattice to enable iterative data-flow analysis:

$$(V, N, R, E, C) \sqcup (V', N', R', E', C') \stackrel{\text{def}}{=} (V \cup V', N \cup N', R \cup R', E \cup E', C \cup'' C').$$

Preserving uniqueness of nodes and edges requires merging two edges with the same start and target nodes. This does not preserve combness in general. However, the following result shows that combness is preserved at control-flow merges during iterative analysis.

[11] It records that the connection between the container object and the new object node satisfies both d and $\diamond_{\mathtt{i}}$. This is the key to proving Lemma 1.

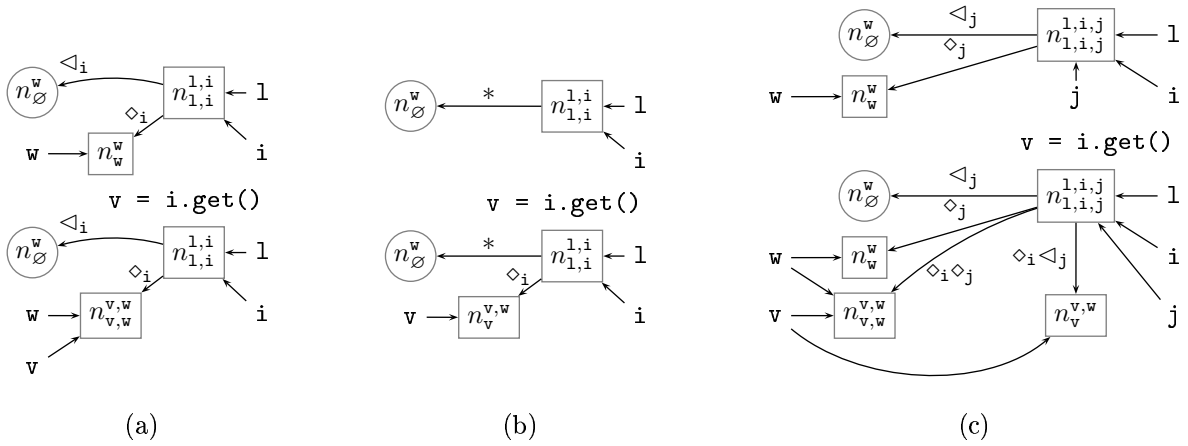


Figure 14: Retrieval through an iterator, comb case

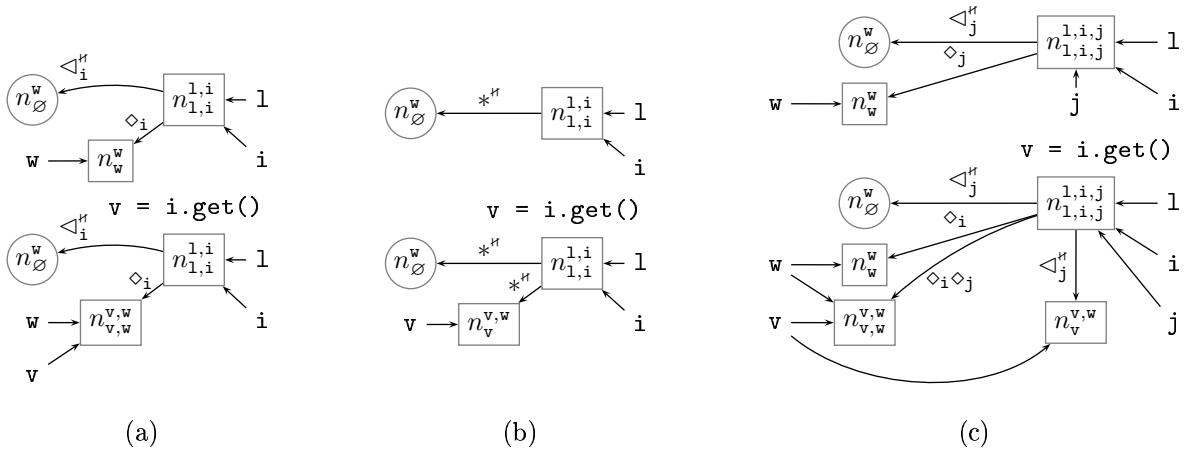


Figure 15: Retrieval through an iterator, non comb case

Theorem 2 *When traversing a merge point in the control-flow graph, it is safe to use operator $\sqcup$ on points-to graphs associated with incoming control-flow edges.*

Proof is given in Section B.7.

5.6 Correctness Issues

We already proved safe handling of directions and correctness of operator $\sqcup$, i.e., most of the novel features introduced in this analysis. Convergence of the iterative analysis is a consequence of points-to graphs bounded size (nodes, edges and directions) and monotonicity of transfer functions. From Theorem 1 and from local safety of each transfer function, we may apply the main result of abstract interpretation (Theorem T2 of [5] page 252) to prove global safety of our pointer analysis. However, because the current abstract interpretation does not consider points-to graph themselves, this correctness proof only deals with properties of individual container edges. Considering $l \xrightarrow{d} o$ in a points-to graph G at program point p, and calling C the set of all possible connections at point p between containers abstracted by l and objects abstracted by o, one has $\alpha(C) \sqsubseteq G$.[12]

[12] Set C can be defined as the join of concrete connections between l and o over all control-flow paths leading to p [15], a.k.a. the collecting semantics at p.

Completing the analysis's proof is left for future work. Establishing the partition property (two nodes may not abstract the same object for a given execution) will probably be a fine point of the proof: the main idea is that new nodes are only created when (1) an object allocation occurs, yielding a distinct heap location, (2) an object node is materialized from a summary node, which implicitly states that some concrete object is "transferred" from the summary node to the object node, and (3) an object node is materialized from another object node (through method `get()`) and both are incompatible (they cannot abstract a heap object at the same time). Sagiv and al. studied a similar issue in [18] and we expect the backbone of their proof to be applicable to our case.

6 Illustrative Examples

Points-to analysis is now applied to small kernels of practical interest: list copy-reversal, element swapping, iterated swapping and nested traversals—a template for many (sparse) linear algebra kernels [4].

6.1 List Copying and Reversal

We apply the iterative algorithm to the list reversal function presented in Section 5.1. It confirms the analysis ability to discriminate current and previous positions of an iterator, using directions.

```
1  static void reverse (Container c, Container d) {
2    Iterator i = c.last ();
3    Iterator j = d.first ();
4    while (!i.atBegin ()) {
5      Object o = i.get ();
6      j.put (o);
7      j.advance ();
8      i.retreat ();
9    }
   }
```

Initially, we assume that variables `c` and `d` are unaliased, that `c` is a comb—hence the comb edge with unknown direction ($*$) to summary node $n^{\varnothing}_{\varnothing}$—and `d` is empty. The points-to analysis for `reverse` is sketched in Figure 16. The new container edge created at the first iteration of point 5 is labeled with $\diamond_{\mathtt{i}}$, because materialization has been applied through a comb edge (see the transfer function for `get()`. At the second iteration of point 5, `o` does not refer an element of `d` because we could check that $\mathsf{Disjoint}(\diamond_{\mathtt{i}}, \triangleright_{\mathtt{i}})$ and we did not materialize from $n^{\circ}_{\varnothing}$. The fixed point is reached after three iterations, and point 9 yields the final points-to graph: it shows that combness of `c` and `d` is preserved.

Now, if we supposed instead that `d` was connected to `c` at the beginning, the result would be quite different. Calling `reverse` on such a configuration would not preserve combness in general, and the points-to analysis detects it, as shown in Figure 17. Combness of `d` is definitively lost at point 6 because $n^{\circ}_{\circ}$ is reachable from two distinct positions in `d`. Notice that a technique which does not allow comparison of container positions would fail to detect combness preservation: it would mix objects extracted from one container with objects stored into the other and the result would be similar to Figure 17.

Eventually, if we supposed that `c` was not a comb container at the beginning, the analysis would safely discover that `d` is not a comb either in the end. As shown in Figure 18, element retrieval still leads to materialization of an object node, but the new container edge is labeled with $*^{\curlywedge}$. Then, we lose disjointness of positions accross iterations: the next element retrieval yields materializations from both $n^{\varnothing}_{\varnothing}$ and $n^{\circ}_{\varnothing}$. This ends up with a container edge labeled $\diamond_{\mathtt{j}} \curlywedge \triangleleft_{\mathtt{j}}$, i.e., one element possibly being attached at several positions in container `d`.

6.2 Swapping Elements

Analysis is applied to a swap function; it shows the node naming scheme in action with non-primitive directions.

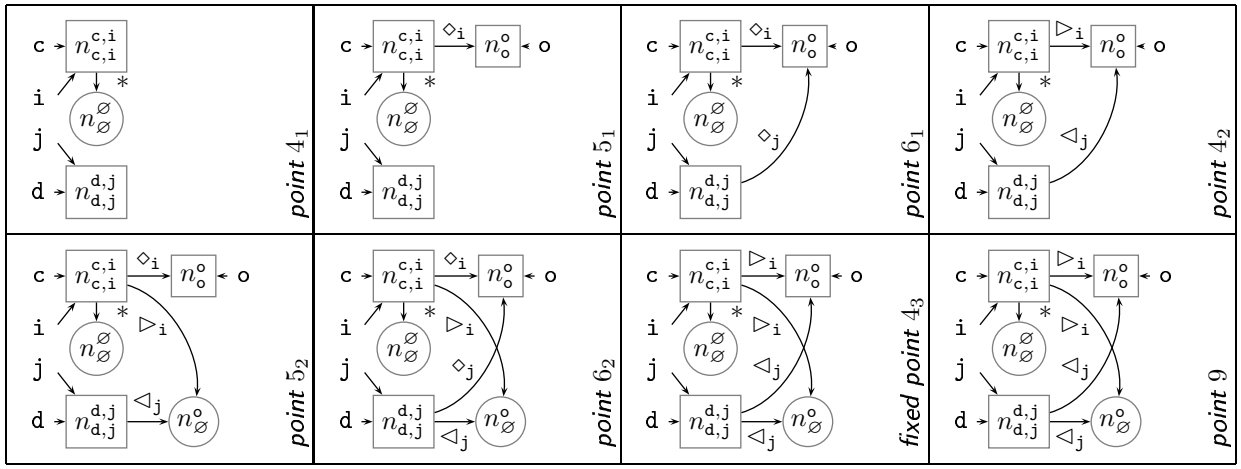


Figure 16: Points-to analysis for `reverse`

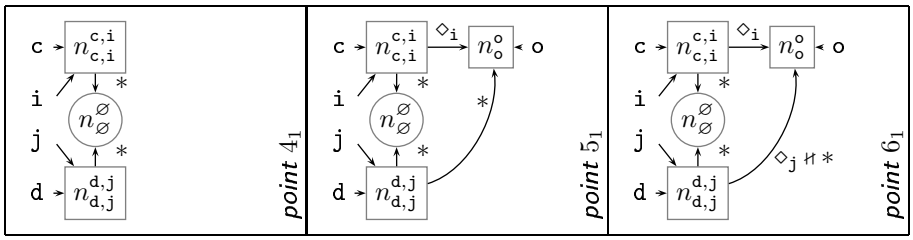


Figure 17: Points-to analysis (first steps) for `reverse` with connected containers

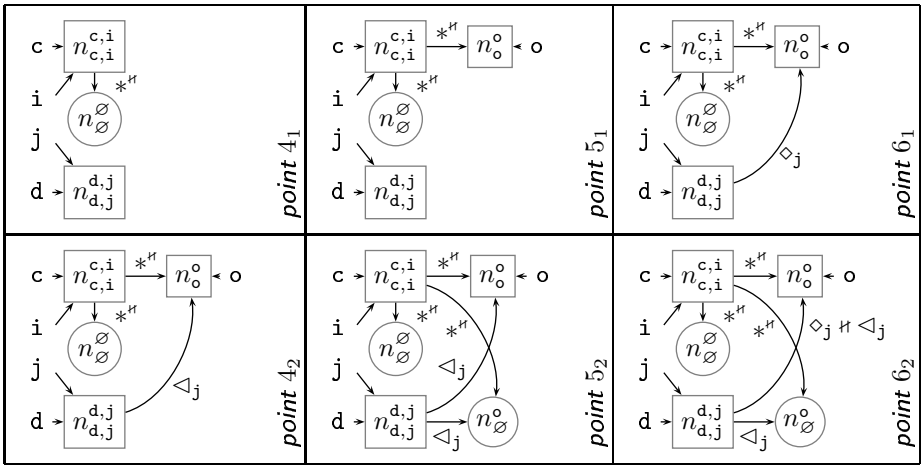


Figure 18: Points-to analysis (first steps) for `reverse` where container `c` is not a comb

```
1  static void swap (Iterator i, Iterator j) {
2    Object a = i.get ();
3    Object b = j.get ();
4    i.put (b);
5    j.put (a);
   }
```

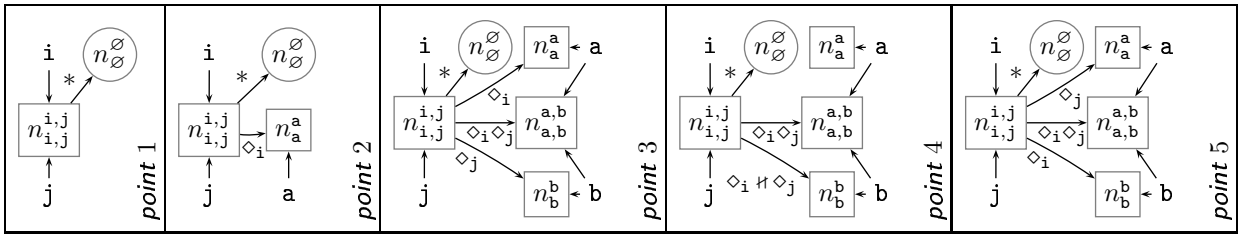


Figure 19: Points-to analysis for `swap`

We suppose that `i` and `j` are iterators attached to the *same comb container*,[13] but positions of `i` and `j` are arbitrary. Analysis for `swap` is sketched in Figure 19. Point 3 demonstrate materialization from both summary and object nodes. Combness of $n^{\mathtt{i,j}}_{\mathtt{i,j}}$ is lost at point 4 because $\diamond_{\mathtt{i}}$ and $\diamond_{\mathtt{j}}$ cannot be compared, but is restored at point 5 thanks to the contextual knowledge captured by directions. Similarly to the destructively reversed list in [18], this example exploits flow-sensitivity and context-aware naming of object nodes.

6.3 In-Place Container Rotation or Reversal

Adding a loop around the previous "swap core" makes things trickier: besides evolution of iterators, an additional difficulty consists in keeping precise knowledge of possible container layouts while using summary nodes. In-place container rotation and reversal are examples of such iterated swapping schemes. They demonstrate three key analysis capabilities:

1. to handle several nodes with distinct history sets (but identical reference sets);
2. to avoid confusions between these nodes thanks to the direction abstraction;
3. and to allow iterative loss and restoration of combness.

Studying the container reversal example, which is very similar to container rotation, we automatically detect combness preservation. We are not aware of any existing technique that achieves such precision on equivalent pointer-chasing container implementations.

```
1  static void reverseInPlace (Container c) {
2    Iterator i = c.first ();
3    Iterator j = c.last ();
4    while (i.isBefore (j)) {
5      Object a = i.get ();
6      Object b = j.get ();
7      i.put (b);
8      j.put (a);
9      i.advance ();
10     j.retreat ();
11   }
   }
```

We suppose that `c` is a comb container. Points-to analysis for `reverseInPlace` is sketched in Figure 20 (variable `c` is not represented). Combness of $n^{\mathtt{i,j}}_{\mathtt{i,j}}$ is lost and restored at each iteration, but eventually preserved.

Notice that history sets are critical for the successful analysis of this example. Using the plain node naming scheme proposed in [18], or using extensions like allocation sites and type properties, we would have to merge nodes $n^{\mathtt{a,b}}_{\mathtt{b}}$ and $n^{\mathtt{b}}_{\mathtt{b}}$ at the second iteration of point 5, as well as $n^{\mathtt{a,b}}_{\varnothing}$ and $n^{\mathtt{b}}_{\varnothing}$ at the second iteration of

[13] Combness is actually the only required information: the case where iterators may *possibly* be attached to different containers can be handled with two aditional object nodes $n^{\mathtt{i}}_{\mathtt{i}}$ and $n^{\mathtt{j}}_{\mathtt{j}}$.

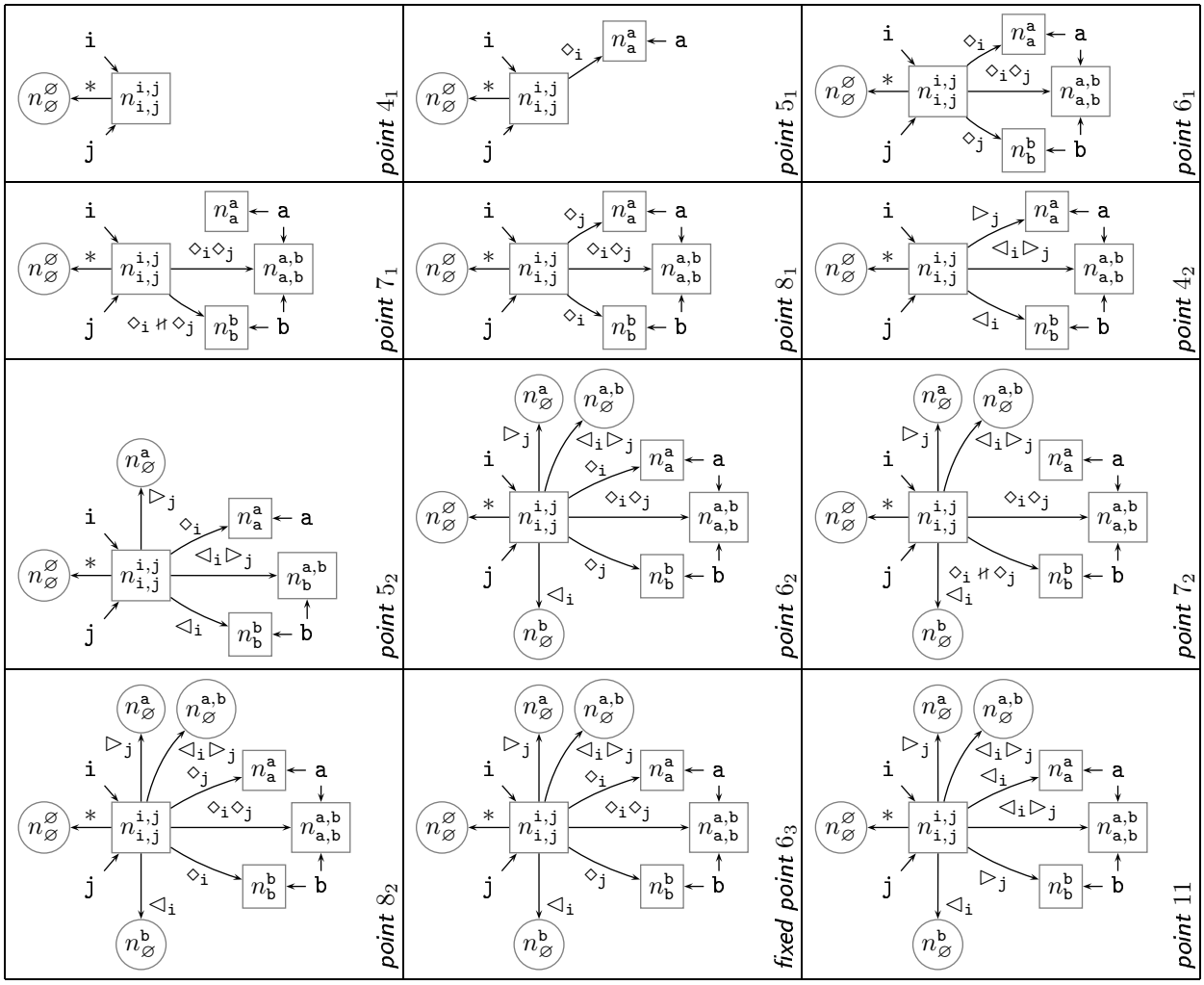


Figure 20: Points-to analysis for `reverseInPlace`

point 6. This would result in merging edges labeled with $\triangleright_j$ and $\triangleleft_i\triangleright_j$, yielding $\triangleleft_i\triangleright_j \amalg \triangleright_j \equiv \triangleleft_i$. Therefore, `j.get()` would conservatively materialize a node $n^{a,b}_{a,b}$ from both n^a_a and n^b_b, with an edge labeled $(\diamond_i \amalg \triangleleft_i)\diamond_j$ instead of $\diamond_i\diamond_j$. This weakens the killing at the second iteration of point 7, and combness is eventually lost at point 8.

6.4 Nested Traversals

Multiple traversals of containers of containers seem to be a very common template for many (sparse) linear algebra kernels. We show that the analysis keeps a reasonable precision when dealing with summarized container nodes. Here is an example with two nested loops traversing a list of lists.

```
1  static void nestedTraversals (Iterator i) {
2    while (!i.atEnd ()) {
3      List l = i.get ();
4      Iterator j = l.first ();
5      while (!j.atEnd ()) {
6        Object o = j.get ();
         j.put (o);
```

```
7
8           j.advance ();
9         }
10        i.advance ();
11      }
    }
```

To decide whether combness is preserved, we assume iterator `i` traverses a comb list of lists abstracted by object $n^{\mathtt{i}}_{\mathtt{i}}$ and the two attached summary nodes $n^{\mathtt{l}}_{\varnothing}$ (abstraction for the second level lists) and $n^{\varnothing}_{\varnothing}$. Full combness of the structure is ensured by combness of edges between $n^{\mathtt{i}}_{\mathtt{i}}$ and $n^{\mathtt{l}}_{\varnothing}$ and between $n^{\mathtt{l}}_{\varnothing}$ and $n^{\varnothing}_{\varnothing}$. The points-to analysis's main steps for `nestedTraversals` are sketched in Figure 21.

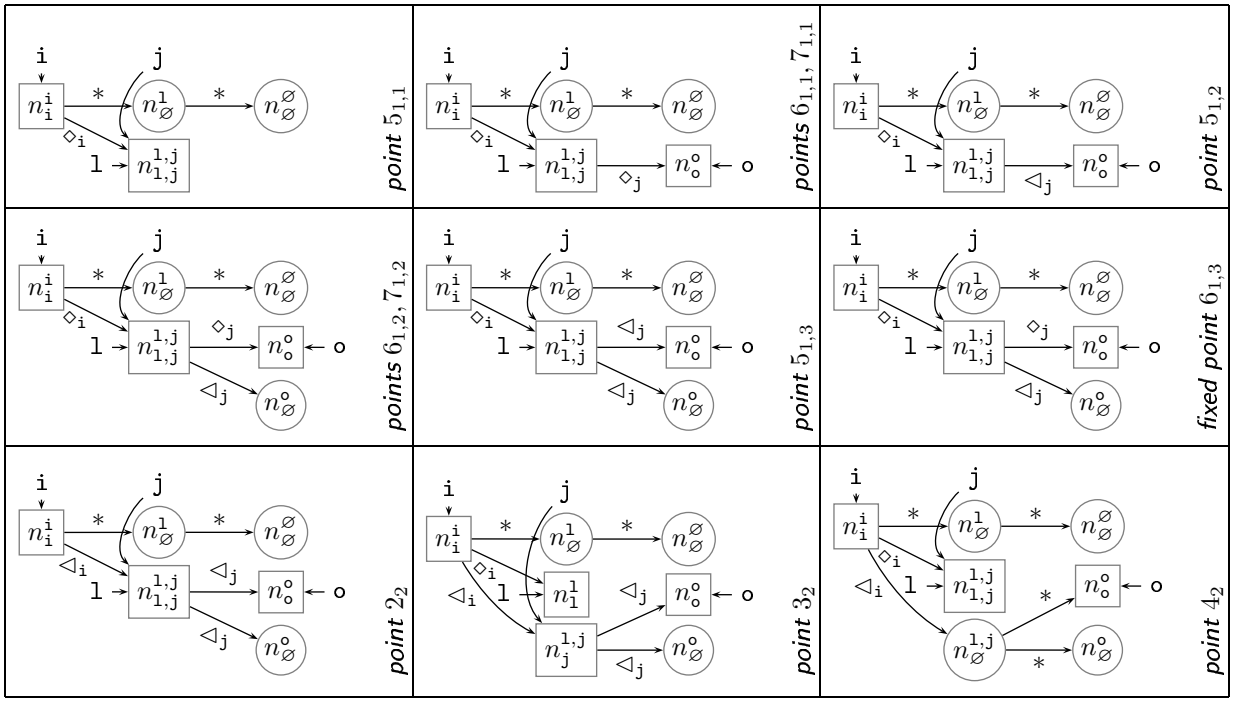


Figure 21: Points-to analysis for `nestedTraversals` (main steps)

7 Aliasing and Combness

Most questions related with static properties of pointers may be solved using points-to graphs.

We overload notation $\longrightarrow$ to denote the *points-to relation* itself (as opposed to points-to graph edges):

$$\forall o, o' \in N: \quad o \longrightarrow o' \overset{\text{def}}{\iff} \big(\exists \mathtt{f} \in \mathsf{Fld} : o \xrightarrow{\mathtt{f}} o' \in E\big) \vee \big(\exists d \in \mathsf{Dir} : o \xrightarrow{d} o' \in C\big)$$
$$\forall \mathtt{v} \in V, o \in N: \quad \mathtt{v} \longrightarrow o \overset{\text{def}}{\iff} \mathtt{v} \longrightarrow o \in R.$$

The transitive closure of the points-to relation is denoted by $\longrightarrow^+$:

$$\forall o, o' \in N: \quad o \longrightarrow^+ o' \overset{\text{def}}{\iff} \exists o_1, \ldots o_k \in N : o \longrightarrow o_1 \wedge o_1 \longrightarrow o_2 \wedge \cdots \wedge o_k \longrightarrow o'$$
$$\forall \mathtt{v} \in V, o \in N: \quad \mathtt{v} \longrightarrow^+ o \overset{\text{def}}{\iff} \exists o_1, \ldots o_k \in N : \mathtt{v} \longrightarrow o_1 \wedge o_1 \longrightarrow o_2 \wedge \cdots \wedge o_k \longrightarrow o.$$

The following equation decides whether two variables are *may-aliases* (i.e., possible aliases); it is defined over $V \times V$:

$$\mathsf{MayAlias}(\mathtt{v}, \mathtt{w}) \iff \big(\mathtt{v} = \mathtt{w}\big) \vee \big(\exists o \in N : \mathtt{v} \longrightarrow o \wedge \mathtt{w} \longrightarrow o\big).$$

In some cases, reference sets in object nodes may help computing *must-aliases*, since a variable v ought to refer some object abstracted by a node n_X^H as soon as $\mathtt{v} \in X$:

$$\mathsf{MustAlias}(\mathtt{v},\mathtt{w}) \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \big(\forall n_X^H \in N : \{\mathtt{v},\mathtt{w}\} \subset X \vee \{\mathtt{v},\mathtt{w}\} \cap X = \varnothing\big).$$

Reachability is the reflexive and transitive closure of the points-to relation, it is defined over $(V \cup N) \times N$:

$$\mathsf{Reach}(o, o') \stackrel{\text{def}}{=} o = o' \vee o \longrightarrow^+ o'.$$

Connectivity is defined over $(V \cup N) \times (V \cup N)$. We take care of not considering paths which traverse incompatible nodes:

$$\begin{aligned}\mathsf{Connect}(o_1, o_1') &\stackrel{\text{def}}{\Longleftrightarrow} o_1 = o_1' \\ &\vee \Big(\exists o_2, \ldots, o_k, o_2', \ldots, o_{k'}' \in N : o_k = o_{k'} \wedge o_1 \longrightarrow o_2 \wedge \cdots \wedge o_{k-1} \longrightarrow o_k \wedge o_1' \longrightarrow o_2' \wedge \cdots \wedge o_{k'-1}' \longrightarrow o_{k'} \\ &\quad \wedge \big(\forall 1 \leq i < k, 1 \leq i' < k' : \mathsf{Compat}(o_i, o_{i'}')\big)\Big).\end{aligned}$$

Combness of individual connections is directly captured within points-to edges, but deciding combness of a full container is more technical (see Figure 1 for a simple example) and several competing definitions exist. A container object (resp. summary) node l abstracts a *comb* container (resp. containers) if two distinct nodes attached to l may not be connected, if each node attached to l is pointed by a comb edge, and if all outgoing edges of nodes *reachable* from l are also comb:

$$\begin{aligned}\mathsf{FullComb}(l) \stackrel{\text{def}}{\Longleftrightarrow} &\neg\big(\exists o, o' \in N, d \in \mathsf{Dir} : \mathsf{Reach}(l, o) \wedge o \xrightarrow{d} o' \in C \wedge \neg\mathsf{Comb}(d)\big) \\ &\wedge \neg\big(\exists o, o' \in N, d, d' \in \mathsf{Dir} : l \xrightarrow{d} o \in C \wedge l \xrightarrow{d'} o' \in C \wedge \mathsf{Connect}(o, o') \wedge \neg\mathsf{Coincide}(d, d')\big).\end{aligned}$$

This definition of combness is very strong, because it relies on predicate Connect. A weaker definition is also useful, it is called *shallow combness* and considers first level pointers only:

$$\mathsf{ShallowComb}(l) \stackrel{\text{def}}{\Longleftrightarrow} \neg\big(\exists o \in N, d \in \mathsf{Dir} : l \xrightarrow{d} o \in C \wedge \neg\mathsf{Comb}(d)\big).$$

Combness and/or shallow combness are critical for many optimizations and (dependence) analyses. Some applications have been discussed in another paper [22] and many other are left for future work.

8 Putting the Analysis to Work

Many important issues have not been covered by the previous formalization and algorithm.

8.1 Cost Improvements

A naming scheme based on sets may induce an exponential number of nodes, but this happens only when many reference assignments are guarded with conditional expressions [18], and when many different iterator variables are attached to the same container. Edges labels can also hold an exponential number of primitive directions, especially when "flattening" directions. Still, it seems that container traversals associated with common algorithms exhibit much regularity, and we believe the size of direction expressions is less likely to grow large than the number of graph nodes.

Nevertheless, some approximations may be necessary to reduce the analysis worst case complexity: bounding the number of nodes via k-limiting on variable sets [18] is one of these. One may also consider approximate simplification rules for directions, either based on k-limiting or specific properties of direction operators. Such heuristics are easy to implement and to prove.

Dealing with standard containers provides a novel and more evolved way to deal with complexity issues. Roughly speaking, we propose a programmer-driven heuristic to switch between Chase-Wegman-Zadeck's efficiency [1]—fast but lacking precision on destructive updates—and Sagiv-Reps-Wilhelm's expensive materialization capabilities—with potentially exponential complexity. Growing variable sets on variable assignments is indeed critical for destructive updating of recursive structures: it enables killing on assignments of

the form `v.f = w` and on `delete()` or `put()` calls. But assuming that typed containers are used, one may choose to precisely kill *container* edges only. Then, variables would be added to reference and history sets for the `get()` method only. This heuristic dramatically reduces points-to graph growth while preserving a good precision. Compared to traditional cost-reduction heuristics which consider all pointers equally important, higher-level objects guide the compiler into more customizable behavior. Our technique unveils practical benefits of high-level (object-oriented) descriptions for static analysis and optimization.

8.2 Precision Improvements

One important feature of [18] has been left out of this presentation because it was of little use in the context of typed container structures, assuming that recursive pointer dereference will be much less frequent. The *sharing* property captures whether an object abstracted by a summary node may be targeted by two separate pointers or not. It is critical for precise materialization from summary nodes with "looping edges" and for shape classification (e.g., acyclic lists, trees, acyclic graphs, cyclic lists, and cyclic graphs). Sagiv and al. define a node predicate named is_shared to capture sharing properties [18]: if $is\_shared(n) = \mathsf{false}$ then a single concrete object abstracted by n at a given run-time instance may only be pointed through one incoming points-to edge. Predicate is_shared has also been extended to avoid confusing multiple fields [4]. Improving our analysis along these lines raises no fundamental problem but makes transformers difficult to understand and does not seem necessary for programs with explicit containers.

Finally, since we used no allocation point in the node naming scheme, an intermediate representation in Static Single Assignment (SSA) [6] would be preferred for the best precision.

9 Related Work

There are several methods addressing the shape and/or alias analysis problems. Some of them are based on program *annotations*, as in Hendren, Hummel and Nicolau [12]. Other methods approximate *access paths* induced by pointers to capture may-alias pairs, like Landi, Ryder and Zhang [16] or Deutsch [8]; Hendren and Nicolau use *path matrices* [13] to classify data structure shapes.

Analyses closer to our work use points-to graphs to capture more general kinds of data structure properties. Some of these are *store-based*—they use allocation sites to name graph—including Chase, Wegman and Zadeck [1] or Jones and Muchnick [14]. The latter uses multiple graphs per program point to cover all possible pointer configurations. Other schemes use k-limiting to bound the number of nodes, including Larus and Hilfinger [17], but usually suffer from exponential growth of the graph size.

Our work is based on Sagiv, Reps and Wilhelm *store-less shape graphs* and *variable set* node naming [18]. It still has a worst case exponential number of nodes but this only happens in tricky conditional pointer assignments. This naming covers multiple pointer configurations in a single graph and capture must-points-to properties for stack variables, critical for precise handling of destructive updating. We extended the naming with *history sets* to improve separation of incompatible pointer configurations in *summary* nodes. Sagiv, Reps and Wilhelm also introduced the *node materialization* scheme to "extract" objects from summary nodes. We generalized it to materialization from *object* nodes (for `get()` operations) allowing simultaneous coverage of multiple container configurations. We showed in Section 8.1 that dealing with explicit container types allowed further reduction of analysis complexity, compared to equivalent pointer-chasing implementations.

Corbera, Asenjo and Zapata [4] proposed additional improvements to shape graphs for the purpose of dependence analysis. Sagiv, Reps and Wilhelm even proposed a general *parametric* shape analysis framework [20] to enable systematic and safe extensions with so-called *instrumentation predicates*. But some of our extensions *cannot* be expressed using instrumentation predicates, e.g., materialization at `get()`. Most of these improvements are compatible and independent from our node naming and container-specific features; they could be useful for further precision and/or cost improvements.

Finally, our experiments established that converting container code to pointer-chasing syntax may dramatically decrease precision; extensions proposed in [18, 4, 20] are of little help, especially in the case of multiple simultaneous container traversals. Other precise techniques like Deutsch's symbolic access paths [8] (based on Parikh mappings and regular expression decompositions) would also fail for lack of absolute naming of iterator positions.

10 Conclusion and Future Work

As programmers look forward to apply object-oriented design models in high-performance applications, compilers have to rely on higher level analyses and optimizations. Pointer analysis for container-centric applications is one of these, and we showed the benefit of using standard libraries and toolkits when exploiting abstract semantics of container and iterator methods. As a tradeoff between efficiency and precision, we proposed natural extensions to previous successful works, introducing combness and dedicated points-to graphs with iterator information.

Theoretical and experimental results confirm the design choices and applicability of our technique. Much work is left for further studies, including complete characterization and proof in abstract interpretation, context-sensitive and modular interprocedural extensions, and evolution of the prototype into a larger compiler infrastructure for real-sized experiments.

It would also be profitable to consider iterator interactions in a more evolved framework: current analysis is rather conservative in the case of iterator cloning. Comparing container positions could involve expressive abstractions such as Presburger arithmetics, especially for arrays [22]. Such approaches seem promising ways to further improve precision; the drawbacks being complexity and serious difficulties to handle structural updates like dynamic insertions and deletions.

Acknowledgments This work is supported by a CNRS-UIUC exchange program. We would like to thank Paul Feautrier for initiating research on combness and exchanging ideas.

References

[1] D. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *ACM Symp. on Principles of Programming Languages (PoPL'93)*, pages 296–310, Charleston, South Carolina, USA, Jan. 1993.

[2] B.-C. Cheng and W.-M. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *ACM Symp. on Programming Language Design and Implementation (PLDI'00)*, June 2000.

[3] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20^{th} ACM Symp. on Principles of Programming Languages (PoPL'93)*, pages 232–245, Charleston, South Carolina, USA, Jan. 1993.

[4] F. Corbera, R. Asenjo, and E. Zapata. New shape analysis techniques for automatic parallelization of c codes. In *ACM Int. Conf. on Supercomputing (ICS'99)*, Rhodes, Greece, June 1999.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4^{th} ACM Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, California, USA, Jan. 1977.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[7] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[8] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *ACM Symp. on Programming Language Design and Implementation (PLDI'94)*, pages 230–241, Orlando, Florida, USA, June 1994.

[9] J. Dolby and A. A. Chien. An evaluation of automatic object inline allocation techniques. In *ACM Symp. on Programming Language Design and Implementation (PLDI'00)*, Vancouver, British Columbia, Canada, June 2000.

[10] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *Int. Journal of Parallel Programming*, 24(6), 1996.

[11] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *23rd ACM Symp. on Principles of Programming Languages (PoPL'96)*, pages 1–15, St. Petersburg Beach, Florida, USA, Jan. 1996.

[12] L. J. Hendren, J. Hummel, , and A. Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. In *ACM Symp. on Programming Language Design and Implementation (PLDI'92)*, pages 249–260, San Francisco, Calfifornia, USA, June 1992.

[13] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):35–47, Jan. 1990.

[14] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *ACM Press*, 1982.

[15] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[16] W. A. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *ACM Symp. on Programming Language Design and Implementation (PLDI'93)*, pages 56–67, Albuquerque, New Mexico, USA, June 1993.

[17] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *ACM Symp. on Programming Language Design and Implementation (PLDI'88)*, pages 21–34, 1988.

[18] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. on Programming Languages and Systems*, 20(1):1–50, Jan. 1998.

[19] S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *23rd ACM Symp. on Principles of Programming Languages (PoPL'96)*, pages 16–31, St. Petersburg Beach, Florida, USA, Jan. 1996.

[20] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *26th ACM Symp. on Principles of Programming Languages (PoPL'99)*, pages 105–118, San Antonio, Texas, USA, Jan. 1999.

[21] R. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *ACM Symp. on Programming Language Design and Implementation (PLDI'95)*, pages 1–12, La Jolla, California, USA, June 1995.

[22] P. Wu, P. Feautrier, D. Padua, and Z. Sura. Points-to analysis for java with applications to loop optimizations. Technical Report CSRD TR-1585, U. of Illinois, Nov. 2000.

[23] P. Wu and D. Padua. Containers, on the parallelization of general-purpose java programs. *Int. Journal of Parallel Programming*, 28(6):589–605, dec 2000.

Appendix

A Working With Directions

One may easily check the following orderings on direction expressions.

$$d'' \sqsubseteq d \shuffle d' \sqsubseteq d \between d' \qquad \varnothing \sqsubseteq d \sqsubseteq d^{\between} \sqsubseteq *^{\between}$$

In addition, $d \sqsubseteq *$ if and only if $\gamma(d)$ holds only *injective* mappings.

Here are the most important simplifications on direction expressions, according to equivalence relation $\equiv$ (proofs of non-trivial results are provided in Section B.2).

- For a given iterator $\mathtt{i}$, primitive direction $\diamond_\mathtt{i}$ only abstracts mappings whose domain is a singleton. This constraint induces strong simplifying properties:

$$\diamond_\mathtt{i}^{\curlywedge} \equiv \diamond_\mathtt{i} \curlywedge \diamond_\mathtt{i} \equiv \diamond_\mathtt{i} \parallel \diamond_\mathtt{i} \equiv \diamond_\mathtt{i} \tag{3}$$

$$\forall d, d' \in \mathsf{Dir} : (\diamond_\mathtt{i} \cdot d) \curlywedge (\diamond_\mathtt{i} \cdot d') = (\diamond_\mathtt{i} \cdot d) \parallel (\diamond_\mathtt{j} \cdot d'). \tag{4}$$

- Primitive directions of an iterator $\mathtt{i}$ partition the set of *comb* connections: they are *exclusive* with respect to operator $\cdot$ and composing them with $\parallel$ yields all comb connections (or all connections when applying unary operator $\curlywedge$).

$$\diamond_\mathtt{i} \cdot \triangleleft_\mathtt{i} \equiv \diamond_\mathtt{i} \cdot \triangleleft_\mathtt{i}^{\curlywedge} \equiv \diamond_\mathtt{i} \cdot \triangleright_\mathtt{i} \equiv \diamond_\mathtt{i} \cdot \triangleright_\mathtt{i}^{\curlywedge} \equiv \triangleleft_\mathtt{i} \cdot \triangleright_\mathtt{i} \equiv \triangleleft_\mathtt{i}^{\curlywedge} \cdot \triangleright_\mathtt{i} \equiv \triangleleft_\mathtt{i} \cdot \triangleright_\mathtt{i}^{\curlywedge} \equiv \triangleleft_\mathtt{i}^{\curlywedge} \cdot \triangleright_\mathtt{i}^{\curlywedge} \equiv \varnothing$$

$$\triangleleft_\mathtt{i} \parallel \diamond_\mathtt{i} \parallel \triangleright_\mathtt{i} \equiv * \qquad (\triangleleft_\mathtt{i} \parallel \diamond_\mathtt{i} \parallel \triangleright_\mathtt{i})^{\curlywedge} \equiv *^{\curlywedge}$$

  As a result, $d \cdot (\triangleleft_\mathtt{i} \parallel \triangleright_\mathtt{i})^{\curlywedge}$ abstract every connection in $\gamma(d)$ whose domain does not contain the current position of i:

$$\gamma(d \cdot (\triangleleft_\mathtt{i} \parallel \triangleright_\mathtt{i})^{\curlywedge}) = \{(\mu, \nu) \in \gamma(d) \mid \nu(\mathtt{i}) \notin \mathsf{Domain}(\mu)\}.$$

- Direction $\varnothing$ is the zero of $\cdot$ and the neutral element of $\parallel$ and $\curlywedge$: $a \parallel \varnothing \equiv a \curlywedge \varnothing \equiv a$ and $a \cdot \varnothing \equiv \varnothing$. Conversely, direction $*^{\curlywedge}$ is the neutral element of $\cdot$ and the zero of $\parallel$ and $\curlywedge$: $a \cdot *^{\curlywedge} \equiv a$ and $a \parallel *^{\curlywedge} \equiv a \curlywedge *^{\curlywedge} \equiv *^{\curlywedge}$.

- Unary operator $\curlywedge$ is idempotent:

$$\forall d \in \mathsf{Dir} : (d^{\curlywedge})^{\curlywedge} \equiv d^{\curlywedge}.$$

  It is *not* the case of binary operator $\curlywedge$.

- Remember $\cdot$ and $\parallel$ are the meet and join operators in Dir. Therefore, considering $b \sqsubseteq a$, $a \cdot b \equiv b$ and $a \parallel b \equiv a$. In particular, $\cdot$ and $\parallel$ are idempotent.

- A weaker property holds for binary operator $\curlywedge$. It states that unary operator $\curlywedge$ is equivalent to iterated application of its binary counterpart.

$$\forall d \in \mathsf{Dir} : \bigcup_{n \geq 1} \underbrace{d \curlywedge d \curlywedge \cdots \curlywedge d}_{n} \equiv d^{\curlywedge}. \tag{5}$$

- Occurrences of $\parallel$ can be replaced by $\curlywedge$ under unary operator $\curlywedge$:

$$\forall a, b \in \mathsf{Dir} : (a \parallel b)^{\curlywedge} \equiv (a \curlywedge b)^{\curlywedge}. \tag{6}$$

- Combness propagates through operator $\cdot$:

$$\forall a, b, c \in \mathsf{Dir} : \mathsf{Comb}(a) \implies a \cdot (b \curlywedge c) \equiv a \cdot (b \parallel c). \tag{7}$$

  As a result, d is a comb direction if and only if it can be expressed without operator $\curlywedge$ (neither binary nor unary versions). An inductive application of (7) gives a *linear-time algorithm* to check any direction expression for combness.

The four operators on directions have distributive and partial distributive properties (proofs in Section B.3).

$$\begin{array}{ll} a \cdot (b \parallel c) \equiv (a \cdot b) \parallel (a \cdot c) & a \parallel (b \cdot c) \equiv (a \parallel b) \cdot (a \parallel c) \\ a \cdot (b \curlywedge c) \equiv (a \cdot b) \curlywedge (a \cdot c) & a \curlywedge (b \cdot c) \equiv (a \curlywedge b) \cdot (a \curlywedge c) \\ a \parallel (b \curlywedge c) \sqsubseteq (a \parallel b) \curlywedge (a \parallel c) & a \curlywedge (b \parallel c) \equiv (a \curlywedge b) \parallel (a \curlywedge c) \\ (a \cdot b)^{\curlywedge} \equiv a^{\curlywedge} \cdot b^{\curlywedge} & (a \curlywedge b)^{\curlywedge} \equiv a^{\curlywedge} \curlywedge b^{\curlywedge} \end{array}$$

Distributivity properties of $\cdot$ and $\parallel$ (resp. $\curlywedge$) are similar to those for logical operators: the lattice of directions is actually distributive. Moreover, (6) states that $(a \parallel b)^{\curlywedge}$ can be simplified into $(a \curlywedge b)^{\curlywedge}$. For implementation and theoretical purposes, this may lead to the definition of *normal forms* for directions. We will only use the weaker "flattening" result, however.

Theorem 3 *Any direction d can be written as an equivalent expression where $\cdot$ only operates on primitive directions $\diamond_{\mathtt{i}}$, $\lhd_{\mathtt{i}}$, $\rhd_{\mathtt{i}}$, and their unary $\curlywedge$ counterparts $\lhd_{\mathtt{i}}^{\curlywedge}$ and $\rhd_{\mathtt{i}}^{\curlywedge}$ (for all iterators $\mathtt{i}$), where $\curlywedge$ operates on such $\cdot$-factors, and where $\shortparallel$ operates on such $\curlywedge$-terms.*

Let us end this section with distributivity properties for α and γ. By definition, γ is distributive over the meet operator of each lattice. Monotonicity only yields a partial result for α:

$$\forall d, d' \in \mathsf{Dir} : \gamma(d \cdot d') = \gamma(d) \cap \gamma(d')$$
$$\forall C, C' \subseteq \mathsf{Conn} : \alpha(C \cap C') \sqsubseteq \alpha(C) \cdot \alpha(C').$$

Conversely, α is distributive over the join operator of each lattice (this is proven in Section B.5). Monotonicity only yields a partial result for γ:

$$\forall C, C' \subseteq \mathsf{Conn} : \alpha(C \cup C') \equiv \alpha(C) \shortparallel \alpha(C')$$
$$\forall d, d' \in \mathsf{Dir} : \gamma(d \shortparallel d') \supseteq \gamma(d) \cup \gamma(d')$$

B Abstract Interpretation Proofs

Two simple properties of γ are used in the proofs.

Lemma 2 *In the definitions of $\gamma(e \shortparallel e')$ and $\gamma(e \curlywedge e')$, one may choose μ_e and μ'_e with disjoint domains without lack of generality.*

Lemma 3 *If (μ, ν) belongs to $\gamma(e)$ and $\mu_{/P}$ is the restriction of μ to a set P of positions (possibly empty), then $(\mu_{/P}, \nu)$ also belongs to $\gamma(e)$.*

B.1 Combness and Comparison

We first review combness properties. All primitive directions abstract only connections (μ, ν) where μ is injective; $\gamma(d \cdot d')$ holds only injective mappings μ if and only if $\gamma(d)$ and/or $\gamma(d')$ do. The disjoint range condition in $\gamma(d \shortparallel d')$ ensures that it holds injective mappings μ when both $\gamma(d)$ and $\gamma(d')$ do, and the condition is also necessary.

Abstract interpretation allows us to give a direct definition of Coincide and Disjoint. It is easy to check that the following equations are equivalent to the inductive definition in Section 3.1.

$$\mathsf{Coincide}(d, d') \iff \forall (\mu, \nu) \in \gamma(d), (\mu', \nu) \in \gamma(d') : \mathsf{Domain}(\mu) \subset \mathsf{Domain}(\mu') \vee \mathsf{Domain}(\mu') \subset \mathsf{Domain}(\mu)$$
$$\mathsf{Disjoint}(d, d') \iff \forall (\mu, \nu) \in \gamma(d), (\mu', \nu) \in \gamma(d') : \mathsf{Domain}(\mu) \cap \mathsf{Domain}(\mu') = \varnothing.$$

Then, it is easy to see that $\mathsf{Coincide}(d, d')$ is equivalent to $d \shortparallel d' \sqsubset \diamond_{\mathtt{i}}$ for some iterator $\mathtt{i}$, and $\mathsf{Disjoint}(d, d')$ is equivalent to $d \cdot d' \equiv \varnothing$.

Remember that coincident directions abstract connections whose domain is either empty or a singleton. This is implied by Lemma 3: μ and μ' with empty or singleton domains are the only kind of mapping which satisfy the definition of Coincide.

B.2 Simplifications

- Equivalence of $*$ and $\lhd_{\mathtt{i}} \shortparallel \diamond_{\mathtt{i}} \shortparallel \rhd_{\mathtt{i}}$.

$$\begin{aligned}\gamma(\lhd_{\mathtt{i}} \shortparallel \diamond_{\mathtt{i}} \shortparallel \rhd_{\mathtt{i}}) = \big\{(\mu, \nu) \in \mathsf{Conn} \mid\ & \exists \mu_1, \mu_2, \mu_3 : \mathsf{Injective}(\mu_1) \wedge \mathsf{Injective}(\mu_3) \\ & \wedge \big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_1(p), \mu_2(p), \mu_3(p)\}\big) \\ & \wedge \big(\forall p \in \mathsf{Domain}(\mu_1) : p < \nu(\mathtt{i})\big) \\ & \wedge \mathsf{Domain}(\mu_2) = \{\nu(\mathtt{i})\} \\ & \wedge \big(\forall p \in \mathsf{Domain}(\mu_3) : p > \nu(\mathtt{i})\big) \\ & \wedge \mathsf{Range}(\mu_1) \cap \mathsf{Range}(\mu_2) = \varnothing \\ & \wedge \mathsf{Range}(\mu_2) \cap \mathsf{Range}(\mu_3) = \varnothing \\ & \wedge \mathsf{Range}(\mu_1) \cap \mathsf{Range}(\mu_3) = \varnothing\big\}.\end{aligned}$$

Thus, the three mappings μ_1, μ_2, μ_3 are required to have disjoint domains, to be injective, and to have disjoint ranges. This is equivalent to μ being injective itself:

$$\gamma(\lhd_{\mathtt{i}} \shortparallel \diamond_{\mathtt{i}} \shortparallel \rhd_{\mathtt{i}}) = \big\{(\mu,\nu) \in \mathsf{Conn} : \mathsf{Injective}(\mu)\big\}.$$

- Iterated application of operator $\nmid$. Let $(\mu,\nu) \in \gamma(d^{\nmid})$.

  $$\exists \mu' : \mathsf{Domain}(\mu') = \mathsf{Domain}(\mu) \wedge (\mu',\nu) \in \gamma(d).$$

  Therefore, applying Lemma 3,

  $$\forall p \in \mathsf{Domain}(\mu), \exists \mu_p : \mu(p) = \mu_p(p) \wedge (\mu_p,\nu) \in \gamma(d).$$

  Calling n the cardinal of $\mathsf{Domain}(\mu)$,

  $$\exists (\mu_k)_{1\le k\le n} : \big(\forall p \in \mathsf{Domain}(\mu), \exists 1 \le k \le n : \mu(p) = \mu_k(p)\big) \wedge \big(\forall 1 \le k \le n : (\mu_k,\nu) \in \gamma(d)\big),$$

  hence $(\mu,\nu) \in \gamma\Big(\bigcup_{n\ge 1} \underbrace{d \nmid d \nmid \cdots \nmid d}_{n}\Big)$. The reciprocal is straightforward.

B.3 Distributivity and Partial Distributivity

Let a, b and c be any directions. We first prove that $a \cdot (b \nmid c) \equiv (a \cdot b) \nmid (a \cdot c)$.

$$\begin{aligned}
(\mu,\nu) \in \gamma(a \cdot (b \nmid c)) \iff & \exists \mu_b, \mu_c : \big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_b(p), \mu_c(p)\}\big) \\
& \wedge (\mu,\nu) \in \gamma(a) \wedge (\mu_b,\nu) \in \gamma(b) \wedge (\mu_c,\nu) \in \gamma(c) \\
\iff & \big(\exists \mu_b, \mu_c : \big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_b(p), \mu_c(p)\}\big) \\
& \wedge (\mu_b,\nu) \in \gamma(a) \cap \gamma(b) \wedge (\mu_c,\nu) \in \gamma(a) \cap \gamma(c) \\
\iff & (\mu,\nu) \in \gamma((a \cdot b) \nmid (a \cdot c)).
\end{aligned}$$

We prove that $a \nmid (b \cdot c) \equiv (a \nmid b) \cdot (a \nmid c)$. Let $(\mu,\nu) \in \gamma(a \nmid (b \cdot c))$. There are mappings μ_a and μ_{bc} such that

$$\big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_a(p), \mu_{bc}(p)\big) \wedge (\mu_a,\nu) \in \gamma(a) \wedge (\mu_{bc} \in \gamma(b) \cap \gamma(c)).$$

Mappings μ_a and μ_{bc} satisfy both conditions of $a \nmid b$ and $a \nmid c$, thus $(\mu,\nu) \in \gamma((a \nmid b) \cdot (a \nmid c))$. Conversely, if $(\mu,\nu) \in \gamma((a \nmid b) \cdot (a \nmid c))$, there are mappings μ_{ab}, μ_{ac}, μ_b, and μ_c such that

$$\begin{aligned}
\big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_{ab}(p), \mu_b(p)\} \wedge \mu(p) \in \{\mu_{ac}(p), \mu_c(p)\}\big) \\
\wedge (\mu_a,\nu) \in \gamma(a) \wedge (\mu'_a,\nu) \in \gamma(a') \wedge (\mu_b \in \gamma(b)) \wedge (\mu_c \in \gamma(c)).
\end{aligned}$$

We can build μ_a such that $\mu_a(p) = \mu_{ab}(p)$ when $\mu(p) = \mu_{ab}(p)$ and $\mu_a(p) = \mu_{ac}(p)$ when $\mu(p) = \mu_{ac}(p)$, and we then replace both μ_{ab} and μ_{ac} in the previous result:

$$\begin{aligned}
\big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_a(p), \mu_b(p)\} \wedge \mu(p) \in \{\mu_a(p), \mu_c(p)\}\big) \\
\wedge (\mu_a,\nu) \in \gamma(a) \wedge (\mu_b \in \gamma(b)) \wedge (\mu_c \in \gamma(c)).
\end{aligned}$$

Now, when $\mu(p)$ differs from $\mu_a(p)$, it has to be equal to both $\mu_b(p)$ and $\mu_c(p)$. We may thus build a mapping μ_{bc} such that

$$\big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_a(p), \mu_{bc}(p)\}\big) \wedge (\mu_a,\nu) \in \gamma(a) \wedge (\mu_{bc} \in \gamma(b) \cap \gamma(c)),$$

hence $(\mu,\nu) \in \gamma(a \nmid (b \cdot c))$.

Proofs for mutual distributivity between $\cdot$ and $\shortparallel$ are likewise: the dijoint ranges condition does not interfere with the former reasoning.

We prove that $a \parallel (b \curlywedge c) \sqsubseteq (a \parallel b) \curlywedge (a \parallel c)$. Let $(\mu, \nu) \in \gamma(a \parallel (b \curlywedge c))$. There are mappings μ_a and μ_b and μ_c such that

$$\big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_a(p), \mu_b(p), \mu_c(p)\}\big) \wedge (\mu_a, \nu) \in \gamma(a) \wedge (\mu_b, \nu) \in \gamma(b) \wedge (\mu_c, \nu) \in \gamma(c) \\ \wedge \mathsf{Range}(\mu_a) \cap (\mathsf{Range}(\mu_b) \cup \mathsf{Range}(\mu_c)) = \varnothing.$$

Since $\mathsf{Range}(\mu_a) \cap (\mathsf{Range}(\mu_b) \cup \mathsf{Range}(\mu_c)) = (\mathsf{Range}(\mu_a) \cap \mathsf{Range}(\mu_b)) \cup (\mathsf{Range}(\mu_a) \cap \mathsf{Range}(\mu_c))$, it shows that $(\mu, \nu) \in \gamma((a \parallel b) \curlywedge (a \parallel c))$.

We eventually prove that $a \curlywedge (b \parallel c) \equiv (a \curlywedge b) \parallel (a \curlywedge c)$. Let $(\mu, \nu) \in \gamma(a \curlywedge b) \parallel (a \curlywedge c))$. There are mappings μ_{ab}, μ_{ac}, μ_b and μ_c such that

$$\big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_{ab}(p), \mu_{ac}(p), \mu_b(p), \mu_c(p)\}\big) \wedge (\mu_{ab}, \nu) \in \gamma(a) \wedge (\mu_{ac}, \nu) \in \gamma(a) \\ \wedge (\mu_b, \nu) \in \gamma(b) \wedge (\mu_c, \nu) \in \gamma(c) \wedge (\mathsf{Range}(\mu_{ab}) \cup \mathsf{Range}(\mu_b)) \cap (\mathsf{Range}(\mu_{ac}) \cup \mathsf{Range}(\mu_c)) = \varnothing.$$

We can build μ_a such that $\mu_a(p) = \mu_{ab}(p)$ when $\mu(p) = \mu_{ab}(p)$ and $\mu_a(p) = \mu_{ac}(p)$ when $\mu(p) = \mu_{ac}(p)$, and we then replace both μ_{ab} and μ_{ac} in the previous result:

$$\big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_a(p), \mu_b(p), \mu_c(p)\}\big) \wedge (\mu_a, \nu) \in \gamma(a) \wedge (\mu_b, \nu) \in \gamma(b) \wedge (\mu_c, \nu) \in \gamma(c) \\ \wedge (\mathsf{Range}(\mu_a) \cup \mathsf{Range}(\mu_b)) \cap (\mathsf{Range}(\mu_a) \cup \mathsf{Range}(\mu_c)) = \varnothing.$$

It shows that $(\mu, \nu) \in \gamma(a \curlywedge (b \parallel c))$. Conversely, let $(\mu, \nu) \in \gamma(a \curlywedge (b \parallel c))$. There are mappings μ_a and μ_b and μ_c such that

$$\big(\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_a(p), \mu_b(p), \mu_c(p)\}\big) \wedge (\mu_a, \nu) \in \gamma(a) \wedge (\mu_b, \nu) \in \gamma(b) \wedge (\mu_c, \nu) \in \gamma(c) \\ \wedge \mathsf{Range}(\mu_b) \cap \mathsf{Range}(\mu_c) = \varnothing.$$

Let μ_{ab} (resp. μ_{ac}) be the restriction of μ_a to positions p where $\mu_a(p) \notin \mathsf{Range}(\mu_c)$ (resp. $\mu_a(p) \notin \mathsf{Range}(\mu_b)$). Since $\mathsf{Range}(\mu_b) \cap \mathsf{Range}(\mu_c) = (\mathsf{Range}(\mu_{ab}) \cup \mathsf{Range}(\mu_b)) \cap (\mathsf{Range}(\mu_{ac}) \cup \mathsf{Range}(\mu_c))$, $(\mu, \nu) \in \gamma(a \curlywedge (b \parallel c))$.

B.4 Lattice Completeness

First of all, we prove completeness for the meet operator. Let e be a *flat* expression associated with some direction d (see Theorem 3). If the *unary* mix operator is not used in e, any *descending* chain starting from e is finite: it traverses sub-expressions (without unary mix operators) only. Now, consider an expression e with unary mix operators. If there is an infinite descending chain starting form e, (5) combined with the previous result enforces that there is an expression e' in that chain with less occurrences of the unary mix operator. An induction on the number of occurrences of the unary mix operator yields a contradication. This proves that every descending chain is finite. Therefore, Dir is a complete meet semi-lattice.

Eventully, Theorem 2.16 in [7] shows that Dir is a complete lattice, because it also has a top element ($*^{\curlywedge}$).

B.5 Galois Connection

Let us prove (2). Consider a set $C \in \mathsf{Conn}$, and call $d = \alpha(C)$. For all $c \in C$, definition of α enforces that $\alpha(c) \sqsubseteq d$, and then $c \in \gamma(d)$ by monotonicity of γ.

Conversely, considering a direction d, there is a direction d' such that $\alpha(\gamma(d)) = \sqsubseteq d$. Monotonicity of γ yields $\gamma(\alpha(\gamma(d))) \subseteq \gamma(d)$. But considering $C = \gamma(d)$ in the result of the previous paragraph yields $\gamma(d) \subseteq \gamma(\alpha(\gamma(d)))$. Therefore $\gamma(d) = \gamma(\alpha(\gamma(d)))$. Injectivity of γ terminates the proof.

Eventually, let us prove that α is distributive over the join operator of each lattice. Consider two sets of connections C and C'. Monotonicity ensures that $\alpha(C \cup C')) \sqsubseteq \alpha(C) \parallel \alpha(C')$. Conversely, let $(\mu, \nu) \in \gamma(\alpha(C) \parallel \alpha(C'))$, and consider μ_1, μ_2 such that $\forall p \in \mathsf{Domain}(\mu) : \mu(p) \in \{\mu_1(p), \mu_2(p)\}$, $(\mu_1, \nu) \in \gamma(\alpha(C))$, $(\mu_2, \nu) \in \gamma(\alpha(C'))$, and $\mathsf{Range}(\mu_1) \cap \mathsf{Range}(\mu_2) = \varnothing$. Using monotonicity again, both (μ_1, ν) and (μ_2, ν) belong to $\gamma(\alpha(C \cup C'))$. Moreover, μ_1 and μ_2 can be choosen with disjoint domains (from Lemma 2). Since μ_1 and μ_2 also have disjoint ranges, a simple inductive reasoning on direction expressions shows that (μ, ν) also belongs to $\gamma(\alpha(C \cup C'))$. Therefore, $\gamma(\alpha(C) \parallel \alpha(C')) \subseteq \gamma(\alpha(C \cup C'))$, which concludes the proof.

B.6 Local Safety

We prove that transfer functions of iterator and container related operations are safe with respect to container connections. This proof relies on the hypothesis that other parts of the analysis—not related to container edges—are correct.

Let (μ, ν) be a connection between a container abstracted by a node l and objects abstracted by a node o. Consider a container edge $l \xrightarrow{d} o$ in a points-to graph G, such that $(\mu, \nu) \in \gamma(d)$. When $\mathsf{Domain}(\mu) = \varnothing$, there might be no such edge, but we do as if there was one labeled with $d = \varnothing$. Consider an operation whose concrete and abstract transfer functions are τ and $\tau^\sharp$, respectively. We show that $\alpha(\tau(\mu, \nu)) \sqsubseteq \tau^\sharp(\alpha(\mu, \nu))$ for methods `first()`, `last()`, `advance()`, `retreat()`, `delete()`, `insert()`, `put()`, and `get()`.

`i=l.first()` (`i=l.last()`). We first consider a container `l` which is not abstracted by node l (the source of the container edge). Concrete transfer function τ is defined as $\tau(\mu, \nu) = (\mu, \nu')$ where $\nu'(\mathtt{i})$ is undefined and $\nu'(\mathtt{j}) = \nu(\mathtt{j})$ for all $\mathtt{j} \neq \mathtt{i}$. Moreover, $\tau^\sharp$ replaces primitive directions of `i` with $*$, hence $\alpha(\mu, \nu') \sqsubseteq \tau^\sharp(\alpha(\mu, \nu')) \equiv \tau^\sharp(\alpha(\mu, \nu))$.

If `l` is abstracted by node l, $\tau(\mu, \nu) = (\mu, \nu'')$ where $\nu'(\mathtt{i}) = 0$ (instead of being undefined) and $\nu'(\mathtt{j}) = \nu(\mathtt{j})$ for all $\mathtt{j} \neq \mathtt{i}$. Because the former definition of ν' is the restriction of ν'' to a smaller set of iterators, one has $\alpha(\mu, \nu'') \sqsubseteq \alpha(\mu, \nu') \sqsubseteq \tau^\sharp(\alpha(\mu, \nu))$.

`i.advance()` (`retreat()`). Concrete semantics of `advance()` states that $\tau(\mu, \nu) = (\mu, \nu')$ where $\nu'(\mathtt{i}) = \nu(\mathtt{i}) + 1$ and $\nu'(\mathtt{j}) = \nu(\mathtt{j})$ for all $\mathtt{j} \neq \mathtt{i}$. Therefore $(\mu, \nu') \in \gamma(\alpha(\mu, \nu)[\lhd_\mathtt{i}/\diamond_\mathtt{i}, */\rhd_\mathtt{i}])$, hence $\alpha(\tau(\mu, \nu)) \sqsubseteq \tau^\sharp(\alpha(\mu, \nu))$.

`i.delete()`. From the concrete semantics of `delete()`, $\tau(\mu, \nu) = (\mu', \nu)$ where $\mu'(\nu(\mathtt{i})$ is undefined and $\mu'(p) = \mu(p)$ for all $p \neq \nu(\mathtt{i})$. As a result, $(\mu', \nu) \in \gamma(\lhd^{\not\pitchfork}_\mathtt{i} \pitchfork \rhd^{\not\pitchfork}_\mathtt{i})$, hence $\alpha(\tau(\mu, \nu)) \sqsubseteq \alpha(\mu, \nu) \cdot (\lhd^{\not\pitchfork}_\mathtt{i} \pitchfork \rhd^{\not\pitchfork}_\mathtt{i}) \equiv \tau^\sharp(\alpha(\mu, \nu))$.

`i.insert(v)`. We suppose that variable `v` references the n-th object created during program execution. From the concrete semantics of `insert()`, $\tau(\mu, \nu) = (\mu', \nu)$ where $\mu'(\nu(\mathtt{i}+1) = n$, $\mu'(p) = \mu(p)$ for all $p \leq \nu(\mathtt{i})$, and $\mu'(p+1) = \mu(p)$ for all $p > \nu(\mathtt{i})$. Of course, (μ, ν) belongs to $\gamma(\alpha(\mu, \nu) \pitchfork \rhd_\mathtt{i})$ as well as (μ'', ν) where μ'' is the restriction of μ' to $\{\nu(\mathtt{i})+1\}$. Therefore, $\alpha(\tau(\mu, \nu)) \sqsubseteq \alpha(\mu, \nu) \pitchfork \rhd^{\not\pitchfork}_\mathtt{i} \equiv \tau^\sharp(\alpha(\mu, \nu))$.

`i.put(v)`. Consider the first rule in the transfer function, where the container edge points to a node o' which does not abstract the object referenced by `v`. Then, the concrete semantics of `put()` coincides with this of `delete()`.

The next two rules are very similar to the `insert()` method. We suppose again that variable `v` references the n-th object created during program execution: $\tau(\mu, \nu) = (\mu', \nu)$ where $\mu'(\nu(\mathtt{i}) = n$ and $\mu'(p) = \mu(p)$ for all $p \neq \nu(\mathtt{i})$. Then, (μ, ν) belongs to $\gamma(\alpha(\mu, \nu) \pitchfork \diamond_\mathtt{i})$ as well as (μ'', ν), where μ'' is the restriction of μ' to $\{\nu(\mathtt{i})\}$. Notice $\alpha(\mu, \nu) \equiv \varnothing$ in the second rule and $\alpha(\mu, \nu) = d$ in the third one. Thus, in both cases, $\alpha(\mu, \nu) \pitchfork \diamond_\mathtt{i} \equiv \tau^\sharp(\alpha(\mu, \nu))$.

Remains the fourth rule, where o, the edge's target, abstracts the object referenced by `v`, and where $\diamond_\mathtt{i} \in d$ (i.e., $\diamond_\mathtt{i}$ appears in every expression of d). The transfer function is $\tau^\sharp(d) = d \pitchfork (d \cdot \diamond_\mathtt{i})$. Once again, $\tau(\mu, \nu) = (\mu', \nu)$ where $\mu'(\nu(\mathtt{i})) = n$ is the creation number associated with the single object abstracted by object node o, and $\mu'(p) = \mu(p)$ for all $p \neq \nu(\mathtt{i})$. From Lemma 1 (proven at the end of this section), there is a connection $(\mu_\mathtt{i}, \nu) \in \gamma(d)$ associated with a real execution of the program which satisfies $\mathsf{Domain}(\mu_\mathtt{i}) = \{\nu(\mathtt{i})\}$. Of course, $\mu_\mathtt{i}$ can be choosen such that $\mu_\mathtt{i}(\nu(\mathtt{i})) = n$. This shows that μ' is partitionned into a mapping $\mu_\mathtt{i}$, and a mapping μ'', defined as the restriction of μ to positions $p \neq \nu(\mathtt{i})$. These two mappings have disjoint domains but their ranges may actually intersect. If they do not intersect, it means that μ'' has empty domain, because there is only one object abstracted by the target node; hence $\mu' = \mu''$ and $(\mu', \nu) \in \gamma(d)$. On the contrary, $(\mu'', \nu) \in \gamma(d)$ and $(\mu_\mathtt{i}, \nu) \in \gamma(d \cdot \diamond_\mathtt{i})$, thus $(\mu', \nu) \in \gamma(d \pitchfork (d \cdot \diamond_\mathtt{i}))$. Finally, $\alpha(\tau(\mu, \nu)) = \alpha(\mu', \nu) \sqsubseteq d \pitchfork (d \cdot \diamond_\mathtt{i}) = \tau^\sharp(d)$.

`v = i.get()`. The first rule (after strong nullification) does not interfere with container edges.

Consider the edge created after materialization in the second and third rules. These rules only differ in the replacement of d by $d \cdot \diamond_\mathtt{i}$ when d is a comb direction. The materialized container edge targets

an object node $o = n^{H \cup \{\mathtt{v}\}}_{X \cup \{\mathtt{v}\}}$, and we call n the single object abstracted by o. From the concrete semantics of `get()`, $\tau(\mu, \nu) = (\mu', \nu)$ where μ' is the restriction of μ to $\{p : \mu(p) = \mu(\nu(\mathtt{i}))\}$. Of course, having $\tau^\sharp(d) = d$ in the non-comb case is safe, because $\mathsf{Domain}(\mu') \subseteq \mathsf{Domain}(\mu)$ and thus $\alpha(\mu', \nu) \sqsubset \alpha(\mu, \nu)$. On the other hand, suppose that d is a comb direction. This ensures injectivity of μ, hence $\mu(p) = \nu(\mathtt{i}) \Longrightarrow p = \nu(\mathtt{i})$. Therefore, the domain of μ' is the singleton $\{\nu(\mathtt{i})\}$. Finally, we have $(\mu', \nu) \in \gamma(d)$ and $(\mu', \nu) \in \gamma(\diamond_\mathtt{i})$, hence $\alpha(\tau(\mu, \nu)) \sqsubseteq \alpha(\mu, \nu) \cdot \diamond_\mathtt{i} \equiv \tau^\sharp(\alpha(\mu, \nu))$.

This section ends with the proof of Lemma 1. We follow the inductive definition of direction expressions, using operators $\cdot$, $\shortparallel$ and $\curlywedge$.

First of all, any connection (μ, ν) such that $\mathsf{Domain}(\mu) = \{\nu(\mathtt{i})\}$ is abstracted by $\diamond_\mathtt{i} \shortparallel d$ and $\diamond_\mathtt{i} \curlywedge d$. Moreover, $(\mu, \nu) \in \gamma(d) \Longrightarrow (\mu, \nu) \in \gamma(d^\curlywedge)$. Eventually, consider a direction of the form $\diamond_\mathtt{i} \cdot d$. As noticed in Section 5.4.3, only two rules—in the transfer functions for `get()` and `put()`—build directions of the form $\diamond_\mathtt{i} \cdot d$. Other graph transformations, including join at control-flow merge points, do not build such directions. Consider a container edge labeled $\diamond_\mathtt{i} \cdot d$ at a program point p following a call to `get()` or `put()`. Let $(\mu', \nu) \in \gamma(\diamond_\mathtt{i} \cdot d)$ be a connection associated with a real execution of the program, i.e., in the collecting semantics at p. From the construction of μ' in the previous safety proofs, $\nu(\mathtt{i})$ belongs to $\mathsf{Domain}(\mu')$. Lemma 3 terminates the proof.

B.7 Merging Edges

We now prove Theorem 2. Let $G = (V, N, R, E, C)$ and $G' = (V', R', N', E', C')$ be two points-to graphs associated with adjacent incoming control-flow edges. Supposing that G and G' are conservative approximations of points-to relations, it must be proven that $G'' = (V'', R'', N'', E'', C'') = G \sqcup G'$ is still a conservative approximation for all paths leading to the merge point through the chosen incoming control-flow edges.

The case of variables, objects and their connections is straightforward, since $V'' = V \cup V'$, $N'' = N \cup N'$, $R'' = R \cup R'$ and $E'' = E \cup E'$.

Let us prove that $C'' = C \cup^{\shortparallel} C'$ is a conservative approximation of container connections after the merge point. Considering in turn conditionals and loops, we show that $e'' = l \xrightarrow{d \shortparallel d'} o$ abstracts every connection previously abstracted by either $e = l \xrightarrow{d} o \in C$ or $e' = l \xrightarrow{d'} o \in C'$.

- Exclusion rules on conditionals (with or without `else` case, and arbitrary `switch` constructs) only allows a single execution path to be valid for a given instance of the merge point during execution. Choosing $d \shortparallel d'$ is thus safe.

- Loops induce two merge points: we consider a given *run-time instance* of either the exit point or the iteration point. Let $(\mu, \nu) \in \gamma(d)$ and $(\mu', \nu) \in \gamma(d')$ be two configurations abstracted by e and e', for this precise run-time instance. Monotonicity properties of the iterative analysis enforce that $\mu \subseteq \mu'$ or $\mu' \subseteq \mu$, i.e., the domain of one mapping includes the other and both mappings coincide on the smaller domain. As a result, joining μ and μ' into a new mapping μ'' yields either $\mu'' = \mu$ or $\mu'' = \mu'$. The mapping μ'' from the configuration $(\mu'', \nu'') \in \gamma(e'')$ at the same run-time instance is thus partitioned into $\mu' \setminus \mu$ and $\mu \setminus \mu'$ with disjoint ranges (one of them is empty). Hence $(\mu'', \nu'') \in \gamma(d \shortparallel d')$.

This concludes the proof, conditionals and loops being the only cases of merge points (no `gotos`).

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399